MINISTERUL ÎNVĂȚĂMÂNTULUI

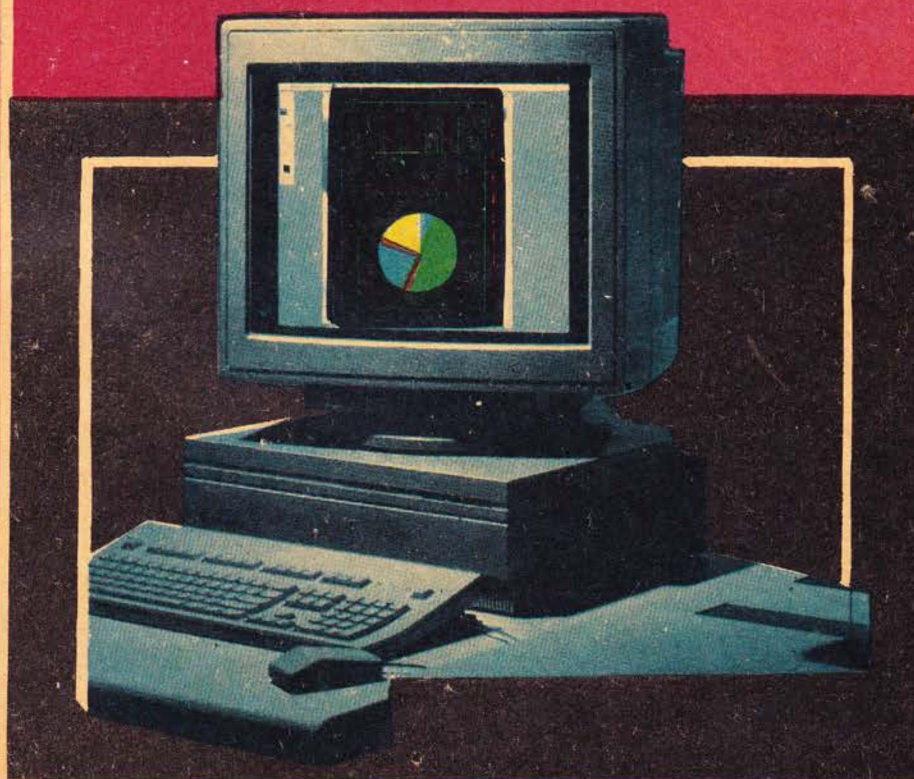
FL. MUNTEANU

T. IONESCU
GH. MUSCĂ

DANIELA TĂTARU
S. M. DASCĂLU

Programarea calculatoarelor

Manual pentru licee de informatică, clasele X-XII



Lei 1318 ISBN 973-30-3162-3

EDITURA DIDACTICĂ ŞI PEDAGOGICĂ, R.A. – BUCUREŞTI, 1994

MINISTERUL ÎNVĂȚĂMÂNTULUI

Conf. dr. ing. **Florin Munteanu**

Coordonator

Prof. dr. ing. **Traian Ionescu**
Ș. l. ing. **Gheorghe Muscă**
Ș. l. ing. **Daniela Tătaru**
Ș. l. ing. **Sergiu Mihai Dascălu**

Programarea calculatoarelor

Manual pentru licee de informatică, clasele X-XII

250

Bogdan Tănasă

clasa a IX-a D



EDITURA DIDACTICĂ ȘI PEDAGOGICĂ, R.A. — BUCUREȘTI

Lucrarea corespunde programei școlare aprobate de Ministerul Învățământului cu nr. 36286/1990 și de Comisia Națională de Informatică cu nr. 987/1990

ISBN 973-30-3162-3

Redactor : ing. *Maria Beluri*
ing. *Marius Murariu*
Tehnoredactor : *Elena Stan*
Coperta : *Elena Drăgulelei*

Coli de tipar : 25,5. B.T. : 14.04-1994
Format 16/70×100. Apărut 1994

Imprimeria „Oltenia" Craiova
B-dul Mareșal Ion Antonescu, 102
Plan 20854/25/1994



CUVÂNT ÎNAINTE

Manualul de față se adresează elevilor claselor a X-a și a XII-a cu profil de informatică, fiind structurat în două părți. Prima parte prezintă limbajul de programare PASCAL, iar cea de-a doua tratează limbajul C, acestea fiind, la ora actuală, cele mai răspândite limbaje de programare.

Manualul urmărește prezentarea detaliată a conceptelor de bază ale celor două limbaje, în versiunea lor standard, însoțind explicarea construcțiilor specifice cu un număr mare de exemple, dificultatea acestora crescând pe parcursul celor două părți ale manualului.

Pentru fixarea cunoștințelor și pentru crearea unor deprinderi de organizare riguroasă a activității de programare, s-a inclus un număr mare de exerciții și probleme, unele fiind însoțite de rezolvări complete, iar altele de răspunsuri sau indicații.

Deși s-a urmărit prezentarea versiunilor standard ale celor două limbaje, autorii recomandă folosirea mediilor integrate de programare Turbo Pascal și, respectiv, Turbo C (Borland) — disponibile pe calculatoare compatibile IBM-PC — acestea asigurând toate facilitățile dezvoltării aplicațiilor software. Sunt oferite astfel posibilități de editare a textului sursă, compilare, link-editare, execuție în regim de depanare, dezvoltare de aplicații modulare și, mai ales, facilitatea de help on-line, instrument foarte util în autoinstruire. De altfel, toate exemplele și problemele rezolvate au fost testate sub mediile de programare Borland.

În ceea ce privește limbajul PASCAL, referința de bază a fost versiunea standard propusă de Kathleen Jensen și Niklaus Wirth în lucrarea „PASCAL, User Manual and Report", iar în ceea ce privește limbajul C, referința de bază a constituit-o standardul ANSI, prezentat de Brian Kernighan și Dennis Ritchie în ediția a doua a lucrării „The C Programming Language".

Această abordare se bazează pe considerentul că versiunile standard asigură universalitatea programelor: un program care folosește numai proceduri și funcții standard se va executa corect pe orice sistem.

Lucrarea cuprinde, pe lângă toate aspectele uzual abordate în prezentarea celor două limbaje, și unele chestiuni de dificultate sporită (structuri

complexe de date, recursivitate, alocare dinamică a memoriei, aritmetica pointerilor etc.), oferind astfel celor interesaţi puncte de pornire în aprofundarea tehnicilor evoluate de programare. În acest sens, recomandăm consultarea bibliografiei furnizate.

Manualul poate fi folosit cu succes şi de către elevii claselor cu alt profil decât informatică, fiind de asemenea util şi studenţilor din primul an de facultate, ca şi tuturor celor ce doresc să devină nu doar simpli utilizatori de programe, ci şi creatori ai acestora.

Autorii

PARTEA I

LIMBAJUL DE PROGRAMARE PASCAL

CAPITOLUL 1

NOŢIUNI INTRODUCTIVE

1.1. HARDWARE ŞI SOFTWARE

În prezent, calculatoarele reprezintă un instrument de lucru din ce în ce mai răspândit. Gama de aplicaţii este extrem de cuprinzătoare, calculatoarele fiind folosite în domenii ca :

— finanţe, comerţ ;
— rezervări hoteliere şi aeriene ;
— industrie (conducerea numerică a maşinilor unelte, conducerea automată a proceselor etc.) ;
— cercetare ştiinţifică (analiza datelor experimentale etc.) ;
— medicină (simulări, analize etc.) ;
— tehnică spaţială ;
— jocuri ;
— industria armamentului ;
— educaţie ;
— sisteme de comunicaţie ;
— tipografii.

Începând din anii '50, industria sistemelor de calcul a cunoscut o dezvoltare rapidă, aceasta accentuându-se în momentul creării circuitelor integrate care au făcut posibilă apariţia calculatoarelor personale.

La început, calculatoarele aveau dimensiuni impresionante, ocupau camere întregi, fiind dispuse pe rafturi metalice şi formate din sute de tuburi cu vid, cuve cu mercur şi panouri de semnalizare optică. Asemănarea cu un magazin de fierărie era atât de puternică încât inginerii de pe atunci vorbeau despre creaţiile lor numindu-le în glumă „**hardware**" (articole de fierărie sau echipament solid). În prezent, un calculator cu o putere de calcul mult mai mare decât în anii '50 poate să încapă destul de uşor într-o servietă, însă principiile ce guvernează operarea sa au rămas în esenţă neschimbate.

Hardware-ul oricărui calculator numeric este format din :

— *procesor* ;
— *memorie* ;
— *dispozitive periferice*.

● **Procesorul** este componenta care *realizează în mod real calculele*. El conţine o **unitate de comandă** care direcţionează operaţiile şi o **unitate aritmetică**, echivalentă cu un calculator electronic (de buzunar), dar mult mai rapidă, capabilă să execute mai mult de 1 milion de operaţii/secundă. Pentru

a putea funcţiona la o asemenea viteză, procesorul trebuie să aibă posibilitatea accesării rapide a datelor, aceasta realizându-se cu ajutorul **memoriei** care înmagazinează aceste date în registrele sau locaţiile sale.

● **Dispozitivele periferice** pot fi descrise prin analogie cu tastele şi afişajul unui calculator electronic de buzunar. Ele permit introducerea datelor în memorie şi afişarea rezultatelor. Cu toate că au o viteză de operare cu mult mai mare decât a omului, dispozitivele periferice sunt lente în comparaţie cu procesorul şi memoria.

● Calculatorului i se pot furniza instrucţiuni prin apăsarea tastelor funcţionale, dar el nu poate fi folosit la o viteză corespunzătoare dacă instrucţiunile nu îi sunt furnizate rapid, pe măsura execuţiei lor. Pentru a face posibil acest lucru, în **memorie**, alături de **date**, se plasează şi **instrucţiunile**, codificate numeric. În funcţionarea sa, calculatorul repetă următorul **ciclu de lucru :**

1. Unitatea centrală extrage din memorie următoarea instrucţiune (pentru a numi această operaţie se utilizează termenul englezesc „fetch").
2. Unitatea de comandă decodifică instrucţiunea şi o transformă în semnale electronice.
3. Ca răspuns la aceste semnale electronice, instrucţiunea va fi executată de către unitatea aritmetică, cu sau fără implicarea memoriei sau a unui dispozitiv periferic.
4. Se reia lucrul de la pasul 1.

În acest mod se pot executa automat, la viteza de lucru a procesorului, chiar şi secvenţe lungi de instrucţiuni, o astfel de secvenţă numindu-se **program**.

Iată câteva **exemple de instrucţiuni simple :**

— citeşte un articol de date de la tastatură şi îl introduce în memorie ;

— copiază un articol de date dintr-o locaţie în alta a memoriei ;

— adună conţinutul a două locaţii de memorie şi depune rezultatul într-o a treia ;

— dacă valoarea conţinută într-o locaţie este negativă, atunci execută o instrucţiune dintr-o altă zonă a programului ; altfel, continuă cu următoarea instrucţiune din şir ;

— trimite un articol de date din memorie către un dispozitiv periferic.

Totalitatea programelor disponibile într-un sistem de calcul constituie aşa-numitul **software.** Acest cuvânt a fost inventat pentru a sublinia faptul că programele sunt la fel de importante ca şi hardware-ul, făcând în acelaşi timp o distincţie netă între aceste două componente care însă nu pot lucra decât împreună.

Una dintre cele mai importante componente ale software-ului este **sistemul de operare,** un set de programe de comandă care se ocupă de rezolvarea multor sarcini necesare în pregătirea şi lansarea în execuţie a programelor utilizator, cum ar fi :

— selectarea programului ce va fi lansat în execuţie ;

— pregătirea datelor ;

— aducerea programelor în memorie ;

— alocarea procesorului etc.

1.2. LIMBAJE DE PROGRAMARE

La început, calculatoarele erau programate cu ajutorul **codului maşină** (adică instrucţiunile le erau furnizate direct în formă numerică), dar foarte repede au fost evidenţiate mai multe neajunsuri ale acestei metode :

— datorită naturii primitive a instrucţiunilor în cod maşină, folosirea acestora era greoaie şi genera multe erori ;

— programele scrise în cod mașină erau prea greu de înțeles și de modificat ;

— programarea reprezenta o activitate consumatoare de timp și costisitoare și din acest motiv era de dorit să se poată folosi un același program pe sisteme de calcul diferite (cerință numită „portabilitate"), ceea ce nu se putea realiza în cazul unui program scris în cod mașină, care era specific unui anumit model de calculator și nu putea fi executat pe nici un altul.

S-a pus atunci problema folosirii unui limbaj natural, cum ar fi limba engleză, dar și această metodă era nepotrivită, așa că, în final, a fost ales un compromis între lizibilitatea și generalitatea limbii engleze și precizia și modul direct de adresare al codului mașină, astfel luând naștere limbajele de programare de nivel înalt.

Limbajele de nivel înalt sînt limbaje independente de calculatorul care le execută programele (spre deosebire de programele de nivel coborât — cod mașină, **limbaje de asamblare** — care sunt specifice tipului de calculator)

Avantajele utilizării limbajelor de nivel înalt în comparație cu cele de asamblare constau în :

— *naturalețe* (apropierea lor de limbajele naturale și/sau limbajul matematic) ;

— *ușurință de înțelegere și utilizare* ;

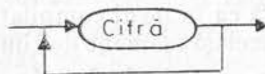
— *portabilitate* (posibilitatea ca același program să fie executat cu modificări minime pe calculatoare de tipuri diferite) ;

— *eficiența în scriere*, datorită facilităților de definire de noi tipuri și structuri de date, operații etc.

Limbajele de nivel înalt, la fel ca oricare alte limbaje, pot fi studiate atât din punctul de vedere al sintaxei (gramaticii) cât și al semanticii (înțelesului). Semantica limbajului Pascal poate fi descrisă ușor cu ajutorul limbii naturale, în timp ce o astfel de abordare a sintaxei ar fi extrem de greoaie. De aceea, pentru descrierea sintaxei limbajului, se folosesc în continuare **diagramele de sintaxă**, definite cu ajutorul unor exemple.

Exemplul 1. În Pascal, definiția unui număr întreg este : un număr întreg reprezintă o secvență de una sau mai multe cifre zecimale. O cifră zecimală este caracterul '0' sau '1' sau ...'9'. Aceleași informații sunt ilustrate în figura 1.1. Diagrama trebuie urmărită de la intrare spre ieșire, cu ajutorul săgeților, memorându-se căile prin care se efectuează trecerea. Atunci când se ajunge la o bifurcație, se poate alege oricare dintre căile care o compun. De exemplu, numărul întreg 365 va putea fi obținut trecând de trei ori prin „cifră zecimală", alegând cea de-a patra, a șaptea și, respectiv, a șasea cale. În conformitate cu figura 1.1., numărul 1,000 nu reprezintă un număr întreg, deoarece diagrama nu conține simbolul ',' (virgulă).

Număr întreg (fără semn):



Cifră zecimală :

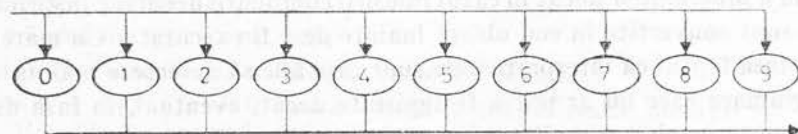


Fig. 1.1.

Exemplul 2. O altă definiţie pe care o putem aborda ca exemplu pentru construirea unei diagrame de sintaxă este aceea a unui **identificator**, noţiune des folosită în cadrul limbajului Pascal pentru a denumi diferite elemente. Descrierea în limbaj natural este: „un identificator reprezintă o secvenţă de caractere care începe cu o literă şi continuă cu niciuna sau cu mai multe litere sau cifre zecimale".

Diagrama de sintaxă asociată este cea din figura 1.2 în care se consideră ca fiind definită „Literă", a cărei diagramă este formată dintr-un număr de căi egal cu numărul literelor alfabetului considerat.

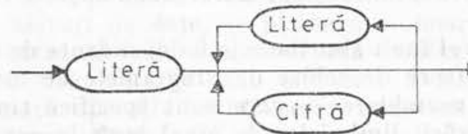


Fig. 1.2.

Un program scris în Pascal (program sursă) nu poate fi executat în mod direct de către hardware-ul unui sistem de calcul, ci trebuie tradus mai întâi într-un set echivalent de instrucţiuni în cod maşină (program obiect), operaţie executată de către un program de sistem (utilitar) numit **compilator**.

Realizarea unui **program scris în Pascal** necesită parcurgerea a **trei etape**:

— *editare* — scrierea programului sursă, cu ajutorul unor programe de sistem (utilitare);

— *compilare* — se aduce în memorie şi se execută compilatorul Pascal. Aceasta determină calculatorul să citească programul sursă, să verifice existenţa posibilelor erori şi să realizeze conversia acestui program în program obiect;

— *execuţie* — programul obiect este adus în memorie şi lansat în execuţie: se efectuează citirea intrărilor, calculele şi scrierea ieşirilor, exact în modul specificat de către programul sursă. Această etapă poate fi repetată ori de câte ori este necesar. Recompilarea se efectuează numai în cazul modificării programului sursă.

Deseori, în timpul compilării sunt semnalate erori de folosire a limbajului, acestea fiind raportate de compilator sub forma unui mesaj de eroare sau a unui număr care se referă la o listă de astfel de mesaje.

În cazul anumitor sisteme de calcul, codul maşină rezultat în urma compilării unui program sursă ar ocupa un spaţiu de memorie inacceptabil de mare în raport cu cel disponibil. De aceea se preferă ca în locul compilatorului să se folosească un **interpretor** care să efectueze la acelaşi moment de timp atât analiza sintactică a instrucţiunii, cât şi execuţia ei. Principalul *dezavantaj* al acestei metode este acela că folosirea interpretorului determină o *execuţie mai lentă a programelor* decât în cazul folosirii compilării deoarece instrucţiunile nu mai sunt convertite în cod obiect înainte de a fi executate. Un mare avantaj este însă faptul că interpretoarele sunt capabile să detecteze mai uşor erori de programare care nu ar putea fi depistate decât, eventual, în faza de execuţie a programelor compilate.

CAPITOLUL 2

PROGRAME. ALGORITMI. ELEMENTE DE PROGRAMARE STRUCTURATĂ

2.1. ETAPELE REALIZĂRII PROGRAMELOR

Procesul de rezolvare a unei probleme începe cu specificarea acesteia şi se încheie cu obţinerea unui program concret şi corect.

Etapele procesului de programare sunt următoarele :

I. *specificarea problemei* ;

II. *găsirea unui algoritm pentru obţinerea soluţiei* ;

III. *codificarea algoritmului într-un limbaj de programare* ;

IV. *testarea şi validarea programului.*

• Etapa I. Specificarea problemei

În prima etapă are loc *analiza problemei.* Rolul analizei constă în elaborarea unui enunţ complet şi precis al problemei, care să ţină seama de condiţiile concrete de realizare şi execuţie a programului. *Enunţul* trebuie să evidenţieze ceea ce urmează să realizeze programul, adică **funcţiile programului.** În acest scop este necesar să se identifice informaţiile de prelucrat (datele de intrare) şi rezultatele cerute (datele de ieşire) ale programului.

Pentru referirea la datele de intrare şi de ieşire se folosesc *variabile de intrare* şi respectiv *de ieşire.* Ele furnizează notaţii simbolice pentru date.

Tot în această etapă se stabilesc reprezentările şi organizarea datelor de intrare şi de ieşire pe suporturile externe de informaţie. Acestea pot fi impuse prin enunţul iniţial al problemei sau pot fi definite de către utilizator.

Rezultatul etapei I este *specificaţia programului.*

Pentru exemplificarea acestei etape se consideră următoarea problemă : „Fiind daţi coeficienţii unei ecuaţii de gradul doi, *a, b* şi *c*, de tip real, să se calculeze (dacă există !) rădăcinile reale ale acesteia. În caz contrar, să se emită un mesaj corespunzător".

1. mărimi de intrare : *a, b, c* ;
   mărimi de ieşire : x1, x2 ;
2. funcţia : calculează rădăcinile reale ale ecuaţiei :
   $a*x^2 + b*x + c = 0$ (dacă există !) sau afişează mesajul :
   „Ecuaţia nu are rădăcini reale" ;

3. organizarea și reprezentarea datelor de intrare și ieșire pe suportul extern : datele se introduc de la tastatură și rezultatele se afișează pe display ;
4. valorile calculate se scriu pe o linie a hîrtiei de imprimantă sub forma : $x_1 = \ldots$ $x_2 = \ldots$

● **Etapa a II-a. Determinarea algoritmului de rezolvare a problemei**

Scopul acestei etape este elaborarea unui algoritm care să realizeze funcțiile programului. Programatorul trebuie să conceapă o *listă de comenzi* care să descrie *secvența de operații* ce va fi executată de către calculator pentru soluționarea problemei. Un calculator devine funcțional dacă este programat adică dacă i se „spune" în cele mai mici amănunte ce să facă. Acest lucru se realizează prin program. În sens general, un program reprezintă descrierea unui algoritm într-o formă interpretabilă („înțeleasă") de către calculator. El rezultă din *codificarea algoritmului într-un limbaj de programare*.

Găsirea algoritmului constituie de cele mai multe ori cea mai grea etapă a procesului programării. Pentru obținerea algoritmului sunt necesare cunoștințe din algebră, analiza matematică, discipline științifice și tehnice. Studiul algoritmilor constituie un domeniu clar delimitat în aria largă a preocupărilor ce constituie știința calculatoarelor. În general, dezvoltarea algoritmului se realizează iterativ, trecându-l prin mai multe niveluri de detaliere. Acest mod de detaliere pas cu pas a specificației este denumită *„proiectare descendentă"*, sugerând faptul că se trece treptat de la o reprezentare generală abstractă, a rezolvării problemei la o reprezentare detaliată a sa.

Problemele simple conduc la un singur algoritm compact, codificat într-o singură unitate de program. În exemplul prezentat anterior algoritmul poate fi explicitat foarte simplu :
Dacă (delta $= b^2 - 4ac$) $\geqslant 0$, atunci calculează

$$x_1 = \frac{-b + \sqrt{\text{delta}}}{2a} \text{ și } x_2 = \frac{-b - \sqrt{\text{delta}}}{2a}$$

altfel afișează mesajul : „Ecuația nu are rădăcini reale".

Problemele complexe și de dimensiuni mari sunt descompuse din punct de vedere logic în *subprobleme* (părți) mai simple, corelate, care pot fi tratate separat, devenind ele însele probleme de rezolvat. Programul rezultat din codificarea unui astfel de algoritm va fi organizat ca un *sistem de module program*. Prin modularizare se simplifică nu numai procesul de dezvoltare și verificare a algoritmului, ci și procesul de codificare, depanare și testare a programelor. Totodată, modularizarea ușurează modificarea programului și deschide posibilitatea refolosirii unităților de program componente.

● **Etapa a III-a. Codificarea algoritmului**

După elaborare, algoritmul este codificat cu ajutorul unui limbaj de programare, obținându-se astfel programul care îl implementează. Limbajul utilizat este ales în conformitate cu specificul problemei, cu particularitățile sistemului de calcul pe care urmează să fie executat programul și, desigur, cu experiența programatorului. Codificarea algoritmului este înlesnită de utilizarea unor simboluri și reguli speciale (organigrame, limbaj pseudocod, diagrame de structură etc.) despre care se va discuta în partea finală a acestui capitol.

● **Etapa a IV-a. Testarea şi validarea programului**

Programul astfel obţinut trebuie *verificat în scopul eliminării erorilor de sintaxă şi al celor de logică.* Chiar dacă în urma execuţiei programului se obţin rezultate (s-au eliminat deci erorile de sintaxă) aceasta nu înseamnă că el este corect, adică realizează funcţiile specificate. Programul poate conţine erori de logică, pentru eliminarea cărora trebuie executat de mai multe ori, folosindu-se seturi de date stabilite pe baza unor criterii considerate ca fiind adecvate problemei.

Activitatea de dezvoltare a unui program nu se încheie în momentul validării programului. Din acel moment începe etapa de întreţinere, care, spre deosebire de celelalte etape, este nelimitată. În etapa de întreţinere sunt efectuate modificări ale programului, fie în scopul corectării unor erori identificate în cursul utilizării sale, fie pentru a-l adapta unor cerinţe noi.

O importanţă deosebită în această etapă o are *documentaţia programului,* care facilitează atât modificările ulterioare ale acestuia, cât şi înţelegerea sa de către alte persoane decât cele care l-au creat.

2.2. ALGORITMI. DEFINIŢIE ; CARACTERISTICI ; REPREZENTARE

2.2.1. Noţiunea de algoritm

Cu toate că algoritmul este o realitate incontestabilă, el nu are în prezent o definiţie riguroasă. Înţelesul mai larg al noţiunii de algoritm este cel de reţetă, metodă, procedeu, dar nu se confundă cu nici unul dintre aceşti termeni.

Se poate spune că un **algoritm** reprezintă o secvenţă finită de operaţii, ordonată şi complet definită, care, pornind de la date (intrări), produce rezultate (ieşiri).

Fiecare propoziţie ce face parte din descrierea unui algoritm este de fapt o comandă care trebuie executată de cineva, de un „calculator" ce poate fi o persoană sau o maşină. Comenzile se adresează calculatorului, care le cunoaşte exact înţelesul. Comanda specifică o operaţie (acţiune) care se aplică datelor algoritmului determinând modificarea acestora. În ansamblu, algoritmul specifică posibile succesiuni de transformări ale datelor, care conduc la aflarea rezultatelor.

Principalele proprietăţi solicitate unui algoritm sunt următoarele :

1. *Să fie bine definit,* adică operaţiile cerute să fie specificate riguros şi fără ambiguitate ;
2. *Să fie descris foarte exact,* astfel încât o maşină programabilă să-l poată realiza ;
3. *Să fie efectiv,* adică să se termine totdeauna după executarea unui număr finit de operaţii ;
4. *Să fie universal,* adică să permită rezolvarea unei clase de probleme.

Observaţii :

1. Un algoritm este o metodă de prelucrare (sau procedură de calcul) deterministă, adică executată la momente diferite de timp conduce în mod necesar la acelaşi rezultat ;

2. Există şi algoritmi nefolosibili în practică, aceştia putând fi foarte lenţi sau necesitând foarte multă memorie.

Cu toate că operațiile de bază (instrucțiunile procesorului) sunt relativ simple, posibilitatea înlănțuirii lor, caracteristică algoritmului, permite realizarea unor prelucrări deosebit de complexe. Iată, de exemplu, algoritmul prin care un calculator care nu cunoaște decât operațiile de adunare, înmulțire și comparare a două valori, poate calcula factorialul lui *n*.

● **Algoritm** (calculul factorialului) :

1. citește valoarea lui n
2. atribuie lui m valoarea 1
3. atribuie lui i valoarea 1
4. atribuie lui m valoarea m*i
5. atribuie lui i valoarea i + 1
6. dacă i <=n atunci treci la 4
7. scrie valoarea lui m
8. stop.

2.2.2. Reprezentarea algoritmilor

Limbajul natural nu permite o descriere suficient de riguroasă a algoritmilor iar, pe de altă parte, odată ce complexitatea problemelor crește, crește și complexitatea descrierilor în limbaj natural. De aceea, pentru reprezentarea algoritmilor se folosesc diferite forme de descriere caracteristice, în fapt *limbajele specializate*.

În general, notația folosită pentru reprezentarea algoritmilor trebuie să satisfacă două cerințe :

1. Să permită exprimarea cât mai naturală a raționamentelor umane, să fie ușor de învățat și de folosit ;

2. Să reflecte caracteristicile limbajelor de programare de nivel înalt pentru a ușura codificarea algoritmilor.

Două dintre cele mai folosite **forme convenționale** de reprezentare a algoritmilor sunt :

— *schemele logice (organigramele)* ;

— *limbajele pseudocod.*

Principala calitate a acestora este posibilitatea de a evidenția cu claritate fluxul controlului algoritmilor (succesiunile posibile ale acțiunilor). Astfel **schemele logice** utilizează în acest scop săgeți de legătură între diferite forme geometrice care simbolizează tipurile de acțiuni, în timp ce **limbajele pseudocod** folosesc cuvinte-cheie, adică niște cuvinte cu înțeles prestabilit ce identifică operația care se execută, și câteva reguli simple de aliniere a textului scris.

● În continuare sunt prezentate blocurile ce pot intra în componența unei **scheme logice (organigramă)** :

1. **Bloc pentru introducerea datelor (citire)**



Fig. 2.1.

unde ListaVariabile cuprinde numele simbolice ale variabilelor cărora li se asociază valori numerice preluate (citite) de pe un suport de informaţie extern.

2. **Bloc de extragere a rezultatelor (scriere)**



Fig. 2.2.

unde variabilele menţionate în listă constituie rezultate ale problemei. Valorile lor sînt preluate din memoria calculatorului şi scrise pe un suport de informaţie extern. Uneori, ca rezultat al unei probleme se poate obţine un text.

3. **Bloc de atribuire**

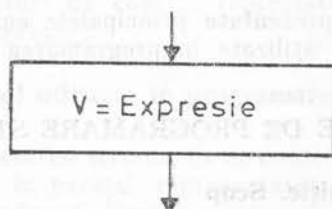


Fig. 2.3.

Un astfel de bloc indică următoarea succesiune de operaţii:

— se calculează expresia din membrul drept;

— se atribuie variabilei din membrul stîng valoarea calculată anterior (V reprezintă numele variabilei).

4. **Bloc de decizie sau bloc de salt condiţionat**

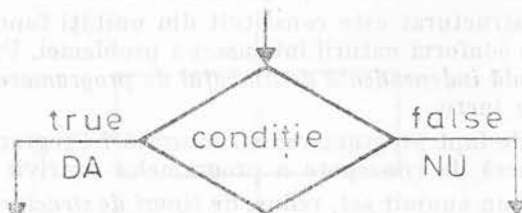


Fig. 2.4.

Condiţia logică înscrisă poate să aibă valoarea „adevărat" sau „fals". În funcţie de valoarea logică obţinută, blocul următor care va fi parcurs va fi legat la ramura „true" (adevărat) sau la ramura „false" (fals).

5. **Bloc de început/sfârșit organigramă**

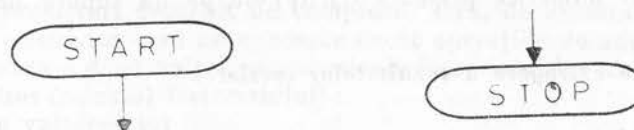


Fig. 2.5.

• **Pseudocodul** permite specificarea algoritmilor cu ajutorul a două tipuri de **enunțuri** : *nestandard* și *standard*. Enunțurile nestandard sunt fraze în limbaj natural care pot fi utilizate de programator în schițarea formelor inițiale ale algoritmilor.

În general, dezvoltarea unui algoritm se realizează iterativ, trecându-l prin mai multe niveluri de detaliere. Proiectarea unui program prin detalierea pas-cu-pas a specificației este denumită *„proiectarea descendentă"*, sugerând faptul că se trece treptat de la o reprezentare generală, abstractă, a rezolvării problemei, la o reprezentare detaliată a sa.

În dezvoltarea algoritmilor, enunțurile nestandard sunt înlocuite treptat cu enunțuri standard, care exprimă operații cu corespondențe directe în limbajele de programare.

În continuare sunt prezentate principalele enunțuri standard corelate cu structurile de control utilizate în programarea structurată.

2.3. ELEMENTE DE PROGRAMARE STRUCTURATĂ

2.3.1. Introducere. Definiție. Scop

La sfârșitul anilor '60, datorită dezvoltării vertiginoase a prelucrărilor de date cu calculatorul, s-au putut aborda și rezolva probleme din ce în ce mai complexe. Programele mari, corespunzătoare acestor probleme s-au complicat în așa măsură încât au devenit foarte greu accesibile chiar și pentru autorii lor. Înțelegerea, depanarea și modificarea unor astfel de programe prezenta dificultăți uneori de neînlăturat. În acea *„criză a software-ului"* s-a ivit, natural, întrebarea : „Se poate elabora o metodologie generală de realizare în mod sistematic, disciplinat a unor programe elegante ?". Ca răspuns la această întrebare s-a cristalizat **metoda programării structurate.**

Un program structurat este constituit din unități funcționale bine conturate, ierarhizate conform naturii intrinseci a problemei. Programarea structurată este o *metodă independentă de limbajul de programare*, ea acționând la nivelul stilului de lucru.

În ce constă de fapt programarea structurată ? Programarea structurată reprezintă o manieră de concepere a programelor potrivit unor reguli bine stabilite, utilizând un anumit set, redus, de *tipuri de structuri de control.*

O structură de control înseamnă o combinație de operații utilizată în scrierea algoritmilor.

Scopul programării structurate este elaborarea unor programe ușor de scris, de depanat și de modificat (actualizat) în caz de necesitate. Programele obținute sunt clare, ordonate, inteligibile, fără salturi și reveniri. Programarea

structurată permite ca programele să poată fi scrise în limbaj pseudocod, limbaj independent de mașină, apropiat de cel natural, convertibil în orice limbaj de programare.

Prin combinarea în mod logic și clar a structurilor de control admise, programarea structurată permite abordarea eficientă a funcțiilor de orice grad de dificultate.

Programarea structurată are la bază o justificare matematică, furnizată de *Boehm* și *Jacopini* și cunoscută ca *„teorema de structură"* care precizează că orice algoritm având o intrare și o ieșire (adică un singur punct de început și un singur punct de terminare a execuției) poate fi reprezentat ca o combinație a trei structuri de control :

1. *Secvența* (succesiune de două sau mai multe operații) ;
2. *Decizia* (alegerea unei operații dintre două alternative posibile) ;
3. *Ciclul cu test inițial* (repetarea unei operații atâta timp cât o anumită condiție este îndeplinită).

Programarea structurată admite și utilizarea altor structuri de control, cum sunt :

4. *Selecția* (permite o alegere între mai mult de două alternative) ;
5. *Ciclul cu test final* ;
6. *Ciclul cu contor.*

Ultimele două structuri de control reprezintă variante ale structurii referită în general ca „iterație".

2.3.2. Structuri de control utilizate în programarea structurată

În continuare, sunt descrise structurile de control utilizate în programarea structurată, folosindu-se, în paralel, reprezentarea cu ajutorul organigramei și reprezentarea prin pseudocod. Pentru a evita posibile confuzii s-au folosit numai notații în limba română (exceptând structura de selecție) deși programatorii experimentați schițează algoritmii în pseudocod folosind termeni din limba engleză, de multe ori aceștia fiind identici cu cuvintele cheie ale limbajului de programare ulterior utilizat. Pseudocodul oferă libertatea folosirii notațiilor considerate ca fiind cele mai sugestive (de exemplu „citește"/„read", „scrie"/„write" etc.).

1. **Secvența.** Reprezintă o succesiune de comenzi care conține o „transformare de date" :

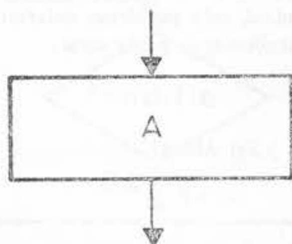


Fig. 2.6.

în care A este o transformare de date. De exemplu :

— atribuire X <———0 sau
M <——— sin(X)

— o secvență de enunțuri nestandard :

citește a, b, c
calculează x1, x2
scrie x1, x2
. . .

2. **Decizia.** Reprezintă alegerea unei operații sau a unei secvențe dintre două alternative posibile.

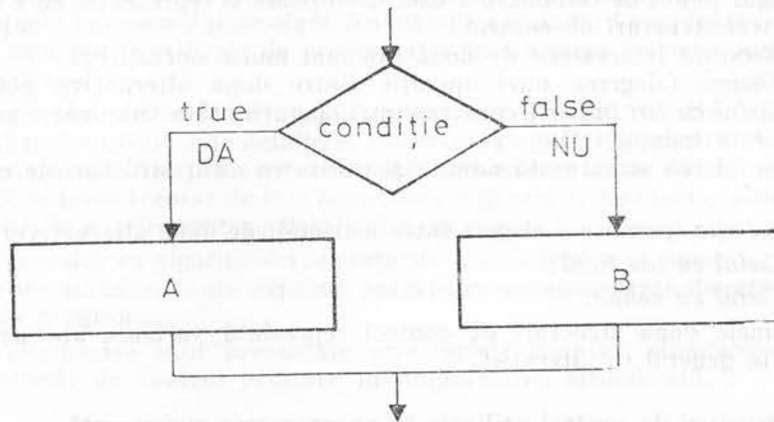


Fig. 2.7.

În limbaj natural, execuția poate fi descrisă astfel :
— se evaluează condiția ;
— dacă rezultatul este adevărat, se execută secvența A ;
— în caz contrar, se execută secvența B.
În pseudocod, execuția se descrie astfel :

```
Dacă condiție Atunci
    secvența A
Altfel
    secvența B
```

Exemplul 1. O problemă banală, în care se impune utilizarea acestei ultime structuri de control, este problema determinării maximului dintre trei numere X, Y, Z. În pseudocod se poate scrie :

```
. . .
Citește X, Y, Z
Dacă ((X > Y) și (X > Z)) Atunci
    max <———X
Altfel
    Dacă (Y > Z) Atunci
        max <——— Y
    Altfel
        max <——— Z
. . .
```

2. bis. **Decizia cu varianta unei căi nule.**

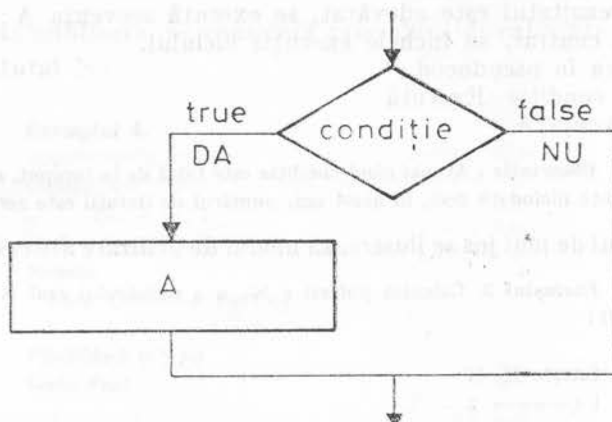


Fig. 2.8.

În limbajul natural, execuţia poate fi descrisă astfel :
— se evaluează condiţia ;
— dacă rezultatul este adevărat, se execută secvenţa A ;
— în caz contrar, se continuă execuţia programului.
În pseudocod, execuţia se descrie astfel :
Dacă condiţie Atunci
    secvenţa A

Exemplul 2. Determinarea valorii absolute a unui număr real x se transcrie cu ajutorul structurii descrise anterior aştfel :
Dacă (x<0) Atunci
  x <— — — —x

3. **Ciclul (bucla, iteraţia).** Asigură executarea unei secvenţe în mod repetat, în funcţie de o anumită condiţie.

>>> *Ciclul cu test iniţial* :

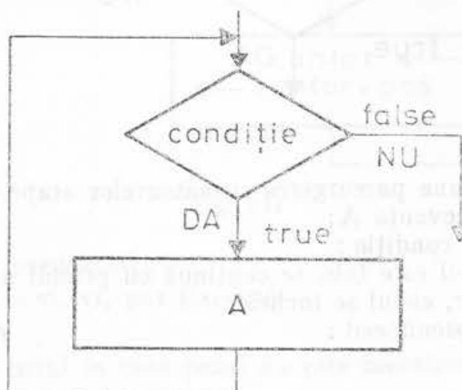


Fig. 2.9.

Execuţia presupune parcurgerea următoarelor etape :
— se evaluează condiţia ;
— dacă rezultatul este adevărat, se execută secvenţa A ;
— în caz contrar, se încheie execuţia ciclului.
Exprimarea în pseudocod :
CâtTimp condiţie Execută
    secvenţa A

Observaţie : Atunci când condiţia este falsă de la început, secvenţa A nu se execută niciodată deci, în acest caz, numărul de iteraţii este zero.

În exemplul de mai jos se ilustrează modul de utilizare al acestei structuri :

Exemplul 3. Calculul puterii a N—a a numărului real X (N, număr natural) :

```
...
    Citeşte X, N
    i <——— 1
    putere <——— 1
    CâtTimp (i <= N) Execută
        putere <——— putere*x
        i <——— i + 1
    Scrie putere
...
```

unde, evident, „putere" va conţine la sfârşit rezultatul dorit.

>>> *Ciclul cu test final* :

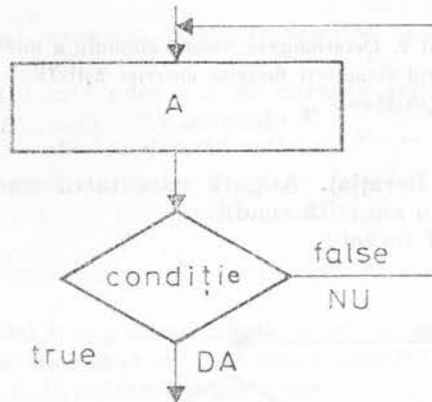


Fig. 2.10.

Execuţia presupune parcurgerea următoarelor etape :
— se execută secvenţa A ;
— se evaluează condiţia ;
— dacă rezultatul este fals, se continuă cu primul pas ;
— în caz contrar, ciclul se încheie.
Exprimarea în pseudocod :
Repetă
    secvenţa A
PânăCând condiţie

Observație : Deoarece testarea condiției se face la sfârșit, secvența se execută cel puțin o dată (numărul de iterații este mai mare decât zero).

Pentru exemplificare, se consideră procedura de calculare a factorialului numărului natural N :

Exemplul 4.

```
Citește N
i    <——— 1
Fact <——— 1
Repetă
  Fact <——— Fact * i
  i    <——— i + 1
PânăCând (i > N)
Scrie Fact
. . . .
```

>>> *Ciclul cu contor :*

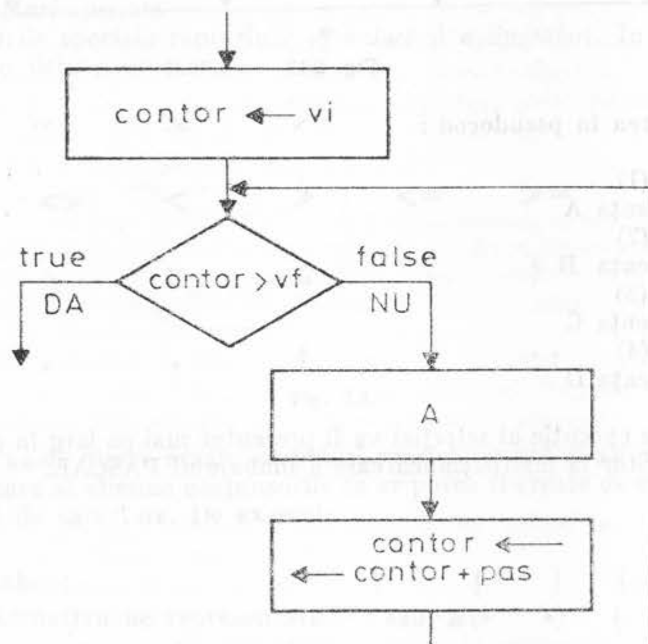


Fig. 2.11.

Exprimarea în pseudocod :

Pentru contor = vi, vf, pas Execută

secvența A

Convenție : În cazul în care pasul nu este menționat, valoarea sa este considerată a fi egală cu 1.

Exemplul 5. Calculul sumei primelor N numere naturale.

```
Citeşte N
Suma <——— 0
Pentru i = 1, N Execută
    Suma <——— Suma + i
Scrie Suma
```

Observaţie : Pentru asigurarea clarităţii programelor, în cazul utilizării pseudocodului, este extrem de importantă alinierea textului („indentarea").

4. **Selecţia.** Reprezintă o extindere a operaţiei de decizie şi permite alegerea unei alternative dintre mai multe posibile.

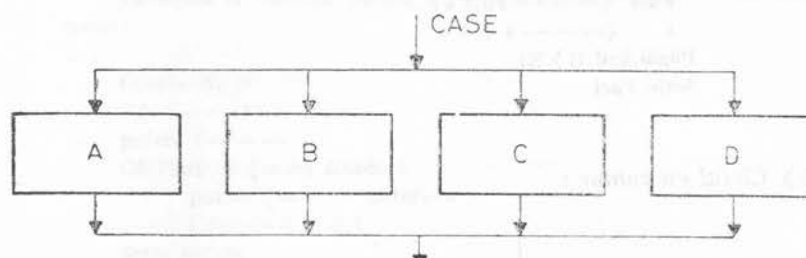


Fig. 2.12.

Exprimarea în pseudocod :

```
do case
    case (1)
     secvenţa A
    case (2)
     secvenţa B
    case (3)
     secvenţa C
    case (4)
     secvenţa D
endcase
```

Modul de execuţie al selecţiei va fi prezentat mai pe larg în cadrul paragrafului referitor la instrucţiunea case a limbajului PASCAL.

CAPITOLUL 3

NOTAȚII ȘI VOCABULAR

Vocabularul de bază al limbajului PASCAL este format din **simboluri de bază**, clasificate în trei mari categorii :

— *litere* — literele alfabetului englez ;
— *cifre* — cifrele arabe : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ;
— *simboluri speciale*.

Simbolurile speciale reprezintă *operatori* și *delimitatori*. În fig. 3.1 este prezentată o listă a acestora.

+	-	*	/					
=	<>	<	>	<=	>=			
(	)	[	]	{	}			
:=	.	,	;	:	..	↑	'	

Fig. 3.1.

Pentru unele dintre aceste simboluri speciale există și alternative de reprezentare care să elimine neajunsurile ce ar putea fi create de structura anumitor seturi de caractere. De exemplu :

simbol :	↑	{	}	[	]
Alternativa de reprezentare :	^ sau @	(*	*)	(.	.)

De exemplu, delimitarea standard a comentariilor se realizează folosind simbolurile '{' și '}', întrucât unele tastaturi nu au taste asociate acestora, s-a prevăzut ca variantă suplimentară utilizarea grupelor de simboluri '(*' și '*)'. Un comentariu este o *porțiune de text cuprinsă între caracterele speciale menționate anterior și care, deși face parte din program, este ignorată complet de către compilator*, rolul său fiind doar de a ușura înțelegerea programului de către cititor.

Tot în categoria simbolurilor speciale se încadrează și **cuvintele rezervate** (cuvintele cheie), care au un înțeles bine stabilit și nu pot fi folosite de către programator decât în contextul permis explicit prin definirea lor în PASCAL. Lista cuvintelor cheie ale limbajului PASCAL standard este următoarea :

and	downot	if	or	then
array	else	in	packed	to
begin	end	label	procedure	type
case	file	mod	program	until
const	for	nil	record	var
div	function	not	repeat	while
do	goto	of	set	with

Identificatorii reprezintă *modalitatea de denumire a constantelor, tipurilor, variabilelor, procedurilor și funcțiilor,* elemente ce vor fi definite ulterior. Un identificator este o secvență de caractere care începe cu o literă ce poate fi urmată de zero sau mai multe litere sau cifre zecimale. Cu toate că definiția nu impune o limitare a numărului de caractere care formează identificatorul, implementarea acestuia necesită precizarea numărului de caractere considerate ca fiind semnificative și care, în varianta standard a limbajului PASCAL este de 8. Aceasta înseamnă că identificatorii care reprezintă obiecte distincte trebuie să difere prin primele 8 caractere. În alte implementări ale limbajului, numărul de caractere semnificative ale unui identificator poate fi mai mare (de exemplu, în Turbo-Pascal 5.0, acesta este 63). *Diagrama de sintaxă pentru identificator* este prezentată în figura 3.2.

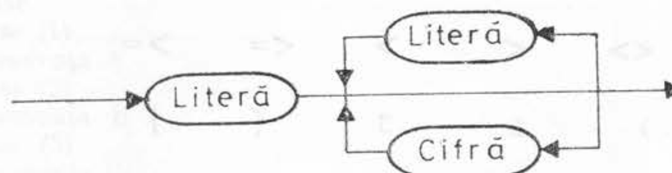


Fig. 3.2.

Observație : Limbajul PASCAL nu face distincție între literele mari și literele mici (nu este „case sensitive"). De exemplu, TOTAL, Total și total au aceeași semnificație pentru program.

- *Exemple de scriere corectă a identificatorilor :*

  suma
  prod4a
  a13bcd

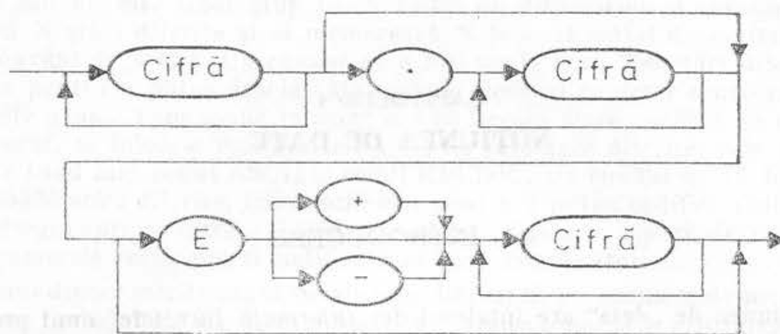
- *Exemple de scriere incorectă a identificatorilor :*

  3rd
  array
  nivel.4
  ceas-1

Există și anumiți identificatori predefiniți, numiți **identificatori standard** sau cuvinte standard (de ex. : cos, sin) și care, spre deosebire de **cuvintele cheie,** pot fi redefiniți de către utilizator, dar această modalitate este rareori folosită în scrierea programelor.

Pentru reprezentarea **numerelor fără semn**, se folosește *notația zecimală*. Litera E, plasată înaintea factorului de scală, are semnificația de „înmulțit cu 10 la puterea". Diagrama de sintaxă pentru numerele fără semn este prezentată în figura 3.3.

Din diagramă rezultă că dacă numărul conține punctul zecimal, acest punct trebuie precedat și succedat de cel puțin câte o cifră (chiar dacă aceasta este zero). De asemenea, în reprezentarea numărului *nu este permisă apariția* **virgulei.**



Fgi. 3.3.

- *Exemple de scriere corectă a numerelor fără semn* :

3 1.0 22728 0.6 5E-4 49.22E+03 1E05

- *Exemple de scriere incorectă a numerelor* :

3,5 XII E10 5.-E6

În cadrul limbajului PASCAL, **spațiile** (blanks), **sfîrșitul de linie** și **comentariile**, sunt considerate ca fiind **separatori.** Între oricare două simboluri PASCAL consecutive este permisă apariția unui număr arbitrar de separatori, cu următoarea excepție : este interzisă apariția separatorilor în cadrul identificatorilor, numerelor sau simbolurilor speciale.

Întrucât numărul separatorilor consecutivi este arbitrar, *limbajul PASCAL are un format liber,* care permite alinierea frazelor din program astfel încât să rezulte un text clar, ușor de urmărit.

Secvențele de caractere cuprinse între două simboluri apostrof se numesc **șiruri.** Pentru a putea include într-un șir caracterul apostrof, se scrie acest caracter de două ori.

- Exemple de șiruri :

'a' ';' '78' 'liceu' 'anii ''60'

' acest șir conține 33 de caractere'

CAPITOLUL 4

NOȚIUNEA DE DATE

4.1. INTRODUCERE

Noțiunea de „**date**" are înțelesul de „*informații furnizate*" unui program sau unei părți a unui program în scopul obținerii anumitor rezultate. Datele pot fi situate în afara calculatorului, caz în care sunt citite prin intermediul unui dispozitiv de intrare sau se pot afla în memorie, fiind rezultatul unor calcule anterioare.

De obicei, în sistemele de calcul, datele se introduc sub formă de **text**, o *secvență de caractere ce aparțin unui anumit set*. Până de curând, textele erau perforate pe cartele (de carton) și erau introduse cu ajutorul unui cititor de cartele. În prezent însă, de cele mai multe ori textele sunt introduse direct în sistemul de calcul, de la un terminal. Există și dispozitive care permit introducerea datelor sub formă de sunete sau imagini, dar acestea sunt mai puțin uzuale și lucrează adeseori cu o conversie a informației tot în formă de text.

Din motive economice, calculatoarele pot prelucra numai un set de caractere fixat și destul de redus, din care fac parte :

— literele ;
— cifrele zecimale ;
— semnele de punctuație uzuale ;
— câteva simboluri matematice.

Cu toate acestea, în calculator poate fi introdusă o foarte largă gamă de date.

O caracteristică importantă a datelor este **tipul** acestora. El stabilește atât *semnificația* datelor cât și *restricțiile* impuse în folosirea lor. **Principalele tipuri de date** ale limbajului PASCAL sunt :

— *Integer* — constituit din numerele întregi ;
— *Boolean* — valorile True (adevărat) și False (fals) ;
— *Real* — numerele cu parte fracționată ;
— *Char* — setul de caractere al sistemului de calcul.

În plus, programatorul are posibilitatea **definirii unor noi tipuri.**

Una dintre caracteristicile limbajelor de nivel înalt este aceea că, în cazul lor, compilatorul realizează o verificare a tipului astfel încât să existe certitudinea că tipurile de date implicate în calcul sunt compatibile cu operațiile ce trebuie realizate.

În afara calculatorului, valorile de orice tip se reprezintă ca texte, adică printr-o secvență de caractere. De exemplu, numerele întregi se pot reprezenta prin secvențe de cifre zecimale, numerele reale prin cifre și punct zecimal, iar caracterele prin ele însele.

În interiorul calculatorului, însă, lucrurile stau cu totul altfel. Hardware-ul unui calculator electronic este format din dispozitive ce se pot afla în două stări fizice distincte. De exemplu, un tranzistor poate conduce sau nu un curent electric. Se spune că un dispozitiv cu două stări memorează o cifră binară sau un **bit.** Unui grup de N astfel de dispozitive îi corespund 2 la puterea N stări diferite și el memorează N biți. O astfel de entitate se va numi **cuvânt de N biți.** Un cuvânt de 8 biți poate avea 256 stări diferite, suficiente pentru a putea asocia câte o stare fiecărui caracter dintr-un set de caractere uzual. Vom spune în acest caz că fiecare stare codifică un caracter. În prezent, se folosesc mai multe coduri de caractere diferite, cele mai răspândite fiind însă codul ASCII și codul EBCDIC. Un cuvânt de 16 biți poate avea 65536 stări diferite, adică suficient pentru a putea codifica toate numerele întregi cuprinse între —32768 și +32767. Celelalte tipuri de date, cum sunt numerele reale, pot fi codificate în mod asemănător.

Unul dintre marile avantaje ale unui limbaj de programare de nivel înalt, cum este limbajul PASCAL, este acela că permite scrierea și utilizarea programelor ca și când calculatorul ar putea gestiona în mod real caractere sau numere și nu șabloane (grupări) de biți.

4.2. CONSTANTE ȘI VARIABILE

Programul poate conține anumite date a căror *valoare nu se modifică*, numite **constante.** Deseori este mai convenabil să se dea un nume simbolic constantei, care să poată fi folosit apoi în program, oriunde este necesară valoarea constantei respective. (De exemplu folosim numele simbolic INCH în loc de constanta 2.54).

Constantele limbajului PASCAL sunt :

— *numerele întregi* ;
— *numerele reale* ;
— *șirurile de caractere* ;
— *constantele desemnate prin identificatori.*

Definițiile primelor trei categorii au fost date în capitolele anterioare, cu ajutorul diagramelor de sintaxă. Pentru ultima categorie, în limbajul PASCAL există următoarele grupe de identificatori care desemnează constante și care se vor prezenta în capitolele următoare :

— identificatorii introduși prin definiții de constante ;
— identificatorii constantelor unui tip enumerare ;
— identificatorii standard **true** și **false** ;
— cuvântul cheie **nil.**

În cadrul programului pot exista și date a căror *valoare trebuie* **modificată.** De exemplu, suma numerelor dintr-un șir dat se modifică ori de câte ori la totalul anterior se mai adaugă un număr. Această categorie de date este modelată în program cu ajutorul **variabilelor** care sunt privite ca entități caracterizate la un moment dat printr-o anumită valoare. Într-un anumit sens, se poate spune că variabila conține valoarea respectivă.

Este posibil însă ca variabila să nu conţină nici o valoare, caz în care se spune că ea este **nedefinită**. Încercarea de a folosi valoarea nedefinită a unei astfel de variabile reprezintă una dintre cele mai uzuale greşeli de programare şi determină o evoluţie impredictibilă a programului : execuţia poate furniza rezultate corecte, poate genera mesaje de eroare sau poate furniza rezultate eronate fără a semnaliza vreo defecţiune.

În cadrul unui program se pot crea şi desfiinţa variabile, valoarea unei variabile nou create fiind nedefinită. Valoarea variabilei rămâne nemodificată atâta timp cât programul nu acţionează asupra ei sau până în momentul în care variabila este desfiinţată.

Variabila este o *entitate căreia i s-a asociat un identificator* (un nume) şi care *poate lua valori corespunzătoare unui anumit tip*. Asocierea identificator (nume) — tip se face printr-o **declaraţie de variabilă**, iar programatorul se va referi la valoarea curentă a variabilei prin numele ei.

Instrucţiunea care modifică valoarea unei variabile se numeşte **instrucţiune de atribuire.**

În general, deosebirea dintre **definiţii şi declaraţii** pe de o parte şi **instrucţiuni** pe de altă parte constă în aceea că prima categorie se foloseşte pentru descrierea tipurilor şi a obiectelor utilizate, în timp ce ultima categorie specifică acţiunile ce trebuie îndeplinite de către calculator în momentul execuţării programului.

În diagrama de sintaxă din figura 4.1 este prezentată definirea unei constante :

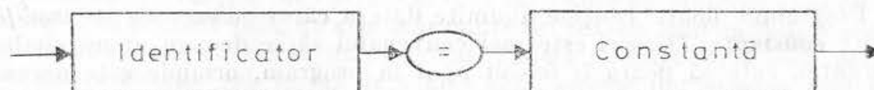


Fig. 4.1

În această diagramă de sintaxă s-a definit prin Identificator numele constantei, iar prin Constantă, valoarea acesteia. Diagrama nu reprezintă sintaxa unei constante ci numai sintaxa definirii acesteia.

Dacă există mai multe definiri de constante, ele trebuie separate cu ajutorul caracterului ';', iar întregul grup va fi introdus prin folosirea cuvântului cheie **const**, ca în exemplul următor :

```
const
    SecInOra = 3600 ;
    AccelGrav = 9.81 ;
    Star = '*'.
```

În acest mod se stabileşte câte un identificator pentru constantele de tip întreg (SecInOra), real (AccelGrav) şi caracter (Star).

Valoarea asociată identificatorului nu se poate modifica prin instrucţiuni, iar tipul său este identic cu al valorii aflate în dreapta semnului '='.

În cazul variabilelor, diagrama de sintaxă a unei declaraţii de variabilă are forma din figura 4.2,

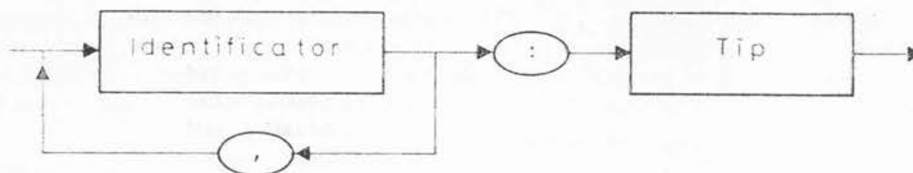


Fig. 3.3.

în care Tip poate fi orice tip admis de limbajul PASCAL.

Partea din program care conţine declaraţii de variabile începe cu cuvântul cheie **var**, urmat de un grup format din una sau mai multe declaraţii de variabile separate prin caracterul ';', ca în exemplul următor :

```
var
    OreÎnZi, ZileÎnAn : integer ;
    Rază, Volum : real ;
    Punct, Virgulă : char ;
    Succes : boolean.
```

În acest mod au fost declarate două variabile de tip întreg (OreÎnZi, ZileÎnAn), două variabile de tip real (Rază, Volum), două variabile de tip caracter (Punct, Virgulă) şi o variabilă de tip boolean (Succes).

Ca o regulă generală a limbajului PASCAL, putem menţiona faptul că **virgula** se foloseşte pentru a separa componentele unei liste **în interiorul :**

— *definiţiilor* ;
— *declaraţiilor* ;
— *instrucţiunilor* ;

în timp ce **între** acestea apare caracterul *punct şi virgulă.*

CAPITOLUL 5

NOŢIUNEA DE TIP. TIPURI STANDARD

Prin **tip** se înţelege o *mulţime de valori căreia i se poate ataşa un nume*; valorile reprezintă constantele de acest tip. În PASCAL există tipuri standard şi tipuri definite de utilizator. În continuare vom prezenta doar tipurile standard, celelalte urmând a fi tratate în unul dintre capitolele următoare.

5.1. TIPUL INTEGER

În PASCAL **tipul integer** permite *reprezentarea, memorarea şi prelucrarea numerelor întregi*. Teoretic, acestea pot avea valori absolute oricât de mari, dar reprezentarea datelor în cadrul calculatorului nu permite existenţa seturilor de numere infinite. Limbajul PASCAL implementează această restricţie furnizând în mod automat un identificator de constantă, **MaxInt**, a cărui valoare reprezintă cel mai mare număr ce poate fi conţinut de o variabilă de tip integer. Valorile tipice ale constantei MaxInt pot fi 32767 pentru microcalculatoare şi 2147483647 pentru sisteme de calcul mari. Practic, tipul integer este format din setul de valori întregi:

—MaxInt, . . ., —1, 0, +1, . . ., +MaxInt.

Observaţie: MaxInt este un exemplu de constantă predefinită; ea poate fi folosită fără ca utilizatorul să fie nevoit a o defini.

Operaţiile aritmetice efectuate asupra valorilor de tip integer vor fi executate corect de către calculator numai dacă toţi operanzii şi toate rezultatele intermediare sunt cuprinse în intervalul —MaxInt până la +MaxInt. Dacă aceste valori sunt depăşite rezultatul devine impredictibil, caz în care se spune că a apărut o „depăşire" (**overflow**) şi programul eşuează. Anumite calculatoare semnalizează o asemenea situaţie generând mesaje de eroare şi determinând oprirea imediată a execuţiei programului. Există însă şi sisteme de calcul în care depăşirile nu sunt detectate, iar programul poate furniza rezultate eronate, fără ca utilizatorul uman să fie atenţionat. Din acest motiv se consideră că eliminarea posibilităţii apariţiei depăşirilor reprezintă o foarte serioasă problemă a programării.

Numerele *de tip* **integer** sunt scrise ca o secvenţă de cifre zecimale, precedate de semnul '—' pentru a indica un număr negativ, de nici un semn sau (opţional) de semnul '+' pentru a indica un număr pozitiv.

Exemplul 1.

```
const
    Nr1 = +2;
    Nr2 = —Nr1;
    Max = MaxInt;
    Min = —MaxInt.
```

Observaţie : Valoarea specificată în cadrul unei definiţii de constantă poate fi exprimată cu ajutorul unui identificator de constantă anterior definit.

● **Operatorii aritmetici** ce se pot aplica asupra valorilor de tip integer sunt :

*	cu semnificaţia de „înmulţire" ;
div	—//— „împărţire şi trunchiere" ;
mod	—//— „modulo"
+	—//— „adunare" ;
—	—//— „scădere".

Operatorul **div** realizează operaţia de împărţire a unui număr întreg prin alt număr întreg şi îndepărtează partea fracţionară a rezultatului, astfel încât operaţia să furnizeze tot un număr întreg.

Exemplul 2.

(+7) div (+2) are ca rezultat +3 iar
(—7) div (+2) are ca rezultat —3.

Încercarea de împărţire a unui număr la zero va genera un mesaj de eroare.

● Operatorul **mod** (prescurtare pentru „modulo") asigură obţinerea ca rezultat a restului împărţirii unui număr întreg prin alt număr întreg. Expresia x **mod** y reprezintă cel mai mic număr care, scăzut din x, dă ca rezultat un multiplu întreg al lui y, deci :

x **mod** y este echivalent cu x—((x **div** y)*y)

Încercarea de a calcula restul în cazul împărţirii unui număr întreg la zero generează mesaj de eroare.

Exemplul 3.

(+18) mod (+5) are ca rezultat +3 iar
(—17) mod (+4) are ca rezultat —1.

Trebuie subliniat faptul că în cazul valorilor de tip integer, înmulţirea şi împărţirea cu acelaşi număr nu reprezintă operaţii complementare deoarece **div** înlătură partea fracţionară. Din acest motiv, ordinea în care se aplică aceşti operatori este deosebit de importantă.

Prioritatea operatorilor aritmetici este următoarea :

1. * div mod
2. + —.

Atunci când este necesar, ordinea de aplicare a acestor operatori în cadrul expresiilor aritmetice poate fi modificată cu ajutorul parantezelor rotunde : '(' şi ')', folosite pe atâtea niveluri cât este necesar. Trebuie precizat însă faptul că utilizarea cumpătată a parantezelor poate clarifica o expresie complicată, în timp ce utilizarea excesivă a acestora poate avea un efect contrar.

Cu ajutorul semnelor '—' şi '+' se poate prefixa semnul primului termen al unei expresii. Semnul '—' reprezintă operatorul de negare: valoarea —x este valoarea lui x cu semn schimbat. Semnul '+' este operatorul identitate: valoarea +x este aceeaşi ca şi x şi de aceea, de obicei, acest semn poate fi omis.

Observaţie: În limbajul PASCAL nu există un operator pentru ridicarea la putere a unui număr. Această situaţie este compensată parţial prin existenţa funcţiei sqr(x) care generează valoarea lui x la pătrat.

● **Funcţiile standard** care lucrează cu valori de tip integer şi generează rezultate de acest tip sunt:

abs(x) — rezultatul reprezintă valoarea absolută a lui x;
sqr(x) — rezultatul reprezintă valoarea lui x la puterea a doua.

Alte funcţii standard care generează rezultate de tip integer:

trunc(x) — rezultatul reprezintă partea întreagă a numărului real x, partea fracţionară fiind îndepărtată;
round(x) — rezultatul reprezintă valoarea numărului real x rotunjită până la următorul întreg.
Adică: pentru x >=0, round(x) înseamnă trunc(x + 0.5)
pentru x < 0, round(x) înseamnă trunc(x — 0.5).

Exemplul 4:

trunc (—3.7)	are ca rezultat	—3
trunc (+3.7)	„	+3
round (—3.8)	„	—4
round (+3.8)	„	+4.

Datorită pericolului apariţiei depăşirilor, nu întotdeauna este de dorit să se rearanjeze expresiile în forma matematică echivalentă.

De exemplu, expresiile (A — B)*(A + B) şi sqr(A) — sqr(B) sunt echivalente din punct de vedere matematic, dar ultima dintre ele poate genera depăşiri în cazul în care A şi B au valori mari.

5.2. TIPUL BOOLEAN (LOGIC)

De cele mai multe ori în timpul executării programului apare necesitatea selectării uneia dintre mai multe acţiuni posibile, în funcţie de anumite condiţii. De asemenea, este posibil să se dorească repetarea unei acţiuni atâta timp cât este satisfăcută o condiţie de continuitate.

În asemenea situaţii, pentru a putea folosi eficient sistemul de calcul, este necesar să dispunem de modalităţi de formulare în program a acestor condiţii pe care calculatorul să le poată testa. Majoritatea condiţiilor se testează uşor: ele sunt sau nu îndeplinite. Se spune că o condiţie are valoarea **true** (adevărat) dacă ea *este îndeplinită*. Altfel, condiţia are valoarea **false** (fals).

Tipul boolean este format din mulţimea valorilor logice **false** *şi* **true**.

Numele „boolean" a fost atribuit acestui tip de valori în memoria matematicianului GEORGE BOOLE (1815—1864), care a pus bazele matematice ale studiului logicii. În cartea sa „Legile gândirii" el a descris algebra valorilor logice, numită astăzi *algebra booleană*, care descrie operarea asupra valorilor logice în acelaşi mod în care algebra convenţională (aritmetică) tratează operarea asupra valorilor întregi.

● **Comparații.** Cel mai adesea, valoarea unei variabile booleene rezultă în urma unei operații de comparație. Valorile comparate pot fi de orice tip dar nu este permisă compararea valorilor de tipuri diferite. Compararea unor valori de tip boolean este o operație mai puțin folosită, dar ea poate furniza uneori formularea elegantă a unor condiții complexe.

În PASCAL există o gamă largă de **operatori de comparație (relaționali)**, dar, deoarece unele dintre simbolurile matematice convenționale nu figurează în setul de caractere al celor mai multe dintre sistemele de calcul, se folosesc următoarele *convenții de notație* :

= cu înțelesul de „egal cu" ;
<> cu înțelesul de „diferit" ;
< cu înțelesul de „mai mic decât" ;
<= cu înțelesul de „mai mic sau egal cu" ;
>= cu înțelesul de „mai mare sau egal cu" ;
> cu înțelesul de „mai mare decât".

Expresiile aritmetice despre care s-a discutat în secțiunea 5.1 fac parte din categoria expresiilor simple. În general, o expresie este fie o expresie simplă, fie o comparație, după cum se arată și în *diagramele de sintaxă* din figura 5.1 :

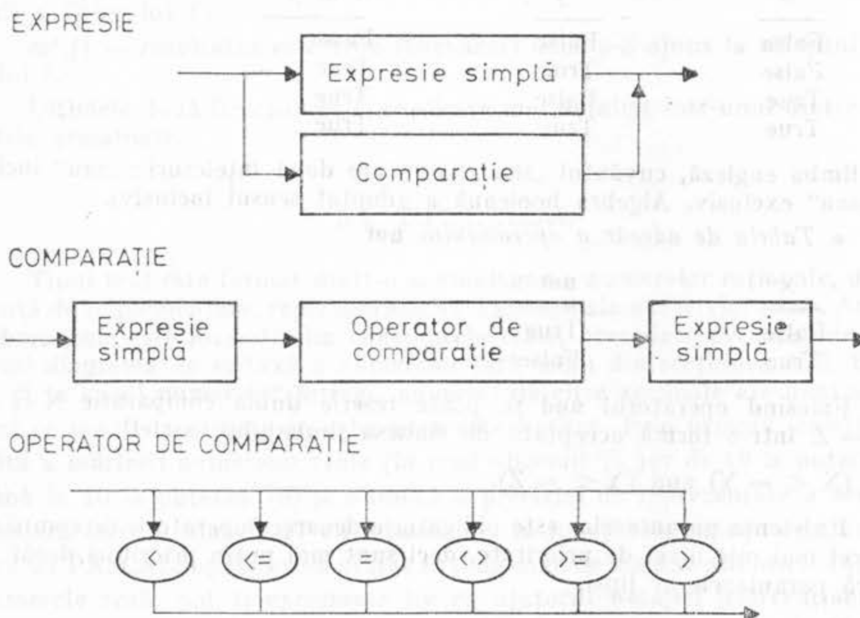


Fig. 5.1.

Această reprezentare generează două consecințe :

1. În PASCAL, operatorii de comparație au cel mai scăzut nivel de prioritate în raport cu toți ceilalți operatori. Expresia 7 – X < Y + Z este echivalentă cu expresia (7 – X) < (Y + Z).

2. Comparațiile duble, cum ar fi X <= Y <= Z, nu sunt permise ca atare și trebuie implementate în alt mod.

Deseori este necesar să construim expresii logice care conţin mai multe comparaţii. Operatorii cu ajutorul cărora putem scrie astfel de expresii se numesc **operatori logici** şi sunt cuvintele-cheie :

and — cu înţelesul de „şi logic" ;
or — cu înţelesul de „sau logic" ;
not — cu înţelesul de „negare logică".

Operanzii folosiţi sunt de tip logic (boolean).

Vom defini funcţiile operatorilor logici folosind „*tabele de adevăr*", în care vom înscrie toate valorile posibile ale operanzilor şi rezultatele corespunzătoare :

• *Tabela de adevăr a operatorului* **and** :

x	y	x **and** y
False	False	False
False	True	False
True	False	False
True	True	True

• *Tabela de adevăr a operatorului* **or** :

x	y	x **or** y
False	False	False
False	True	True
True	False	True
True	True	True

(în limba engleză, cuvântul „sau" („or") are două înţelesuri : „sau" inclusiv şi „sau" exclusiv. Algebra booleană a adoptat sensul inclusiv).

• *Tabela de adevăr a operatorului* **not**

x	**not** x
False	True
True	False

Folosind operatorul **and** se poate rescrie dubla comparaţie X < = Y < = Z într-o formă acceptată de sintaxa limbajului, astfel :

(X < = Y) and (Y < = Z).

Existenţa parantezelor este obligatorie deoarece operatorii de comparaţie au cel mai mic nivel de prioritate, deci sunt mai puţin prioritari decât **and.** Dacă parantezele ar lipsi,

X < = Y and Y < = Z

ar fi echivalentă cu

X < = (Y and Y) < = Z

o expresie fără sens, deoarece operanzii cărora li se aplică **and** trebuie să fie de tip logic şi deoarece sintaxa nu permite o dublă comparaţie.

Pentru a preîntâmpina apariţia unor astfel de erori, în PASCAL se închid între paranteze toate comparaţiile folosite ca operanzi ai operaţiilor logice **and**, **or** şi **not**.

Datele de tip logic pot fi folosite în mod asemănător celor de tip integer. Se pot defini constante de tip logic, se pot calcula expresii de tip logic, iar valoarea rezultată se poate atribui unei variabile de tip logic şi, de asemenea, se pot afişa valori de tip logic. Există însă o restricţie în limbajul PASCAL : valorile de tip logic nu pot fi citite (cu instrucţiunea read).

Observaţie : În limbajul PASCAL există relaţii de ordonare pentru toate tipurile de variabile, inclusiv pentru tipul logic. Se consideră că :
false < true.

Prioritatea operatorilor logici este următoarea :

1. not
2. and
3. or

Ordinea de aplicare a acestor operatori poate fi modificată cu ajutorul parantezelor.

● **Funcţiile standard** care furnizează rezultate de tip logic sunt :

odd(x) — rezultatul este true (adevărat) dacă x este un întreg impar ; altfel rezultatul este false (fals) ;

eoln(f) — rezultatul este true (adevărat) dacă s-a ajuns la sfârşitul unei linii a fişierului f ;

eof(f) — rezultatul este true (adevărat) dacă s-a ajuns la sfârşitul fişierului f.

Ultimele două funcţii vor fi explicate mai detaliat într-unul dintre capitolele următoare.

5.3. TIPUL REAL

Tipul real este format dintr-o submulţime a numerelor raţionale, dependentă de implementare, reprezentând aproximaţii ale numerelor reale. Această submulţime este formată din constantele reale, precedate eventual de semn (vezi diagrama de sintaxă a numerelor fără semn din secţiunea 4.3). La fel ca şi în cazul numerelor întregi, numărul de cifre zecimale ale unui număr real ce pot fi memorate în calculator este limitat. Prin urmare, există o limită a **mărimii** numerelor reale (în mod obişnuit în jur de 10 la puterea 30 până la 10 la puterea 70) şi o limită a **preciziei** de reprezentare a acestora (variază între 6 pînă la 17 cifre zecimale, în funcţie de sistemul de calcul).

În PASCAL, aşa cum s-a arătat în diagrama de sintaxă amintită anterior, numerele reale pot fi exprimate fie cu ajutorul **notaţiei** [**convenţionale** a fracţiilor zecimale, fie cu ajutorul **notaţiei ştiinţifice**, care permite scrierea unor numere foarte mari sau foarte mici.

Exemplul 5.

47E10 este echivalent cu 470000000000.0
—8.5e3 este echivalent cu —8500.0
6E-4 este echivalent cu 0.0006

Caracterul 'E' (sau 'e') se citeşte ca „înmulţit cu 10 la puterea". Expresiile de tip real se compun în mod asemănător cu cele de tip întreg.

Operatorii ce pot fi folosiți sunt următorii:

+ — cu înțelesul de „adunare";

— — cu înțelesul de „scădere";

* — cu înțelesul de „înmulțire";

/ — cu înțelesul de „împărțire de tip real".

Operația de „împărțire de tip real" furnizează aproximarea cât mai corectă a câtului împărțirii a două numere reale. Această operație se poate aplica și în cazul împărțirii a două numere întregi, furnizând ca rezultat un număr real. (A se compara cu rezultatul operației de împărțire de tip întreg, **div**).

Prioritatea operatorilor folosiți în expresii de *tip real* :

1. * /
2. + —

● **Funcții standard** care acceptă un *parametru de tip integer sau real și furnizează rezultatele de același tip* :

abs(x) — calculează valoarea absolută a lui x ;

sqr(x) — calculează pătratul lui x.

● **Funcții standard** care *acceptă un parametru de tip real și furnizează rezultate de același tip* :

sin (x) — calculează sin (x) ;
cos (x) — calculează cos (x) ;
arctan (x) — calculează arctg (x) ;
exp (x) — calculează valoarea e la puterea x ;
ln (x) — calculează logaritmul lui x în baza e ;
sqrt (x) — calculează rădăcina pătrată a lui x.

Observație : Parametrii funcțiilor sin și cos ca și rezultatul funcției arctan sunt unghiuri exprimate în radiani.

● **Funcții standard** care *acceptă un singur parametru de tip real și furnizează rezultate de tip integer* :

round (x) — calculează valoarea lui x rotunjită la cel mai apropiat întreg ;

trunc (x) — calculează valoarea lui x trunchiată la partea întreagă.

Exemplul 6.

round (2.6) are ca rezultat valoarea 3
trunc (2.6) are ca rezultat valoarea 2
round (—2.6) are ca rezultat valoarea —3
trunc (—2.6) are ca rezultat valoarea —2

Expresiile de tip real pot fi comparate cu ajutorul oricărora dintre operatorii relaționali (de comparație). Operatorii aritmetici și unele funcții standard reale pot provoca depășire flotantă.

● **Reguli ce se aplică în cazul tipului Real :**

1. Deoarece fiecare număr întreg are un echivalent număr real, expresiile de tip real pot conține operanzi de tip integer (care sunt convertiți în mod automat la tipul real).

2. Este permisă atribuirea de valori întregi variabilelor de tip real.
3. Operanzii de tip real nu pot fi folosiţi în expresii de tip integer, iar atribuirea de valori reale variabilelor de tip întreg nu este permisă.

Observaţie : Datorită aproximării numerelor în cadrul operaţiilor aritmetice de tip real, egalităţi matematice evidente, cum ar fi :
x/y*y = x sau x/y/z*y = x/z
nu sunt întotdeauna adevărate. Verificaţi, de exemplu, ultima relaţie pentru cazul x = 9.631, y = 4.777 şi z = 3. În timpul scrierii programelor care lucrează cu date de tip real este necesar să avem permanent în vedere posibilele erori generate de aceste aproximări în reprezentarea internă a numerelor.

5.4. TIPUL CHAR (CARACTER)

Datele caracter reprezintă cel mai obişnuit mod de comunicare între oameni şi între sistemele de calcul. O valoare de tip caracter este un element al unui set finit şi ordonat de caractere. Cele mai uzuale seturi de caractere sunt : ASCII (sau ISO) care conţine 128 caractere şi EBCDIC care conţine 256 caractere. Toate seturile de caractere includ litere, cifre zecimale, caracterul spaţiu (blank), unele semne de punctuaţie şi anumite simboluri matematice.

Un caracter cuprins între apostrofuri reprezintă o constantă de tip char (vezi secţiunea 4.2). Pentru a reprezenta caracterul apostrof, acesta se scrie de două ori.

Observaţie : Convenţia de a închide caracterele între apostrofuri se foloseşte numai în cadrul textului unui program PASCAL, fiind necesară pentru a se putea face distincţie între valoarea caracter şi simbolurile folosite în program. Ea nu se foloseşte însă în timpul execuţiei programului la introducerea datelor de tip caracter.

În cadrul unui sistem de calcul, ordinea alfabetică este extinsă la întregul set de caractere. Din păcate, diferitele seturi de caractere au diferite moduri de ordonare. De exemplu : în cazul setului ASCII, cifrele zecimale apar înaintea literelor în timp ce în cazul setului EBCDIC ordinea este inversată. Cu toate acesta, în toate sistemele de calcul se respectă ordinea alfabetică : 'A' < 'B', 'B' < 'C' etc. şi ordinea numerică a cifrelor : '0' < '1', '1' < '2' etc. În cadrul acestei ordonări, fiecărui caracter îi corespunde un număr întreg, numit **număr de ordine.**

● **Funcţiile standard** care permit vizualizarea acestor corespondenţe (directă şi inversă) se mai numesc şi *funcţii de transfer* şi sunt următoarele :
ord (c) — furnizează ca rezultat numărul de ordine al caracterului c în cadrul setului de caractere ;
chr (i) — furnizează ca rezultat valoarea caracterului cu numărul de ordine i.

Observaţie : ord şi chr sunt funcţii inverse : chr(ord(c)) are ca rezultat caracterul c şi ord (chr(i)) produce rezultatul i, numărul de ordine al caracterului. Mai mult, ordonarea unui set de caractere este definită prin :
c1 < c2 dacă şi numai dacă ord(c1) < ord(c2).
Această definiţie poate fi extinsă pentru oricare dintre operatorii relaţionali : =, < >, <, <=, >=, >. Dacă notăm cu R unul dintre aceşti operatori, atunci :
c1 R c2 dacă şi numai dacă ord(c1) R ord(c2)

● **Alte funcții standard** ce pot avea argument de tip char :

pred (c) — funcția predecesor ; furnizează ca rezultat caracterul ce precede caracterul c în setul dat ;

succ (c) — funcția succesor ; furnizează ca rezultat caracterul ce succede caracterul c în setul dat.

● **Prioritățile de aplicare a operatorilor**

În cadrul acestei secțiuni vom prezenta prioritatea tuturor operatorilor amintiți anterior, subliniind faptul că ordinea lor de aplicare poate fi modificată prin utilizarea parantezelor rotunde.

Nivelurile de prioritate au fost notate cu cifre, priorității celei mai mari asociindu-i-se valoarea 1, iar priorității celei mai mici valoarea 4 :

1. not
2. * / and div mod
3. + — or
4. = <> <= >= <>.

EXERCIȚII

1. Ce operatori aritmetici se pot aplica datelor de tip integer ? Care este ordinea de prioritate a acestora ?

2. Precizați funcțiile standard asociate tipului de date integer.

3. Ce operatori relaționali cunoașteți ?

4. Desenați diagrama de sintaxă asociată numerelor reale.

5. Scrieți în notație zecimală obișnuită următoarele numere :

45.67E—4	981E—2	3.00E0
1.885E+1	2.82743E+3	93E3

6. Enumerați funcțiile standard asociate tipului de date real.

7. Considerând expresiile :

a. pred(succ(pred('E')))

b. chr(ord('b') + 3)

să se precizeze care este valoarea acestora.

8. Enumerați tipurile standard de date existente în limbajul PASCAL.

9. Care dintre expresiile următoare sunt corecte și care nu sunt corecte ? Motivați răspunsul. Precizare : x, y, z sunt de tip integer.

a. x < y + z ;

b. x < y or z < y.

10. Ce valoare logică are expresia a **div** b * b = a în care a și b sunt variabile de tip integer, cu b diferit de 0 ?

11. Specificați ordinea operațiilor în evaluarea expresiilor :

a. (ord(car) + 1)*5/a + 2 ;

b. **not** (a + 1 > b*5) **and** t **or** (c < b),

în care a, b, c sunt întregi, t — variabila booleană iar car variabilă de tip char.

12. Să se scrie o expresie logică (booleană) care primește valoarea true dacă un caracter, c, este o cifră.

13. Scrieți o expresie logică a cărei valoare să fie TRUE dacă un număr întreg A este multiplu unui alt număr întreg B și FALSE în caz ocntrar.

14. Scrieți o expresie logică a cărei valoare să fie TRUE dacă o dată calendaristică reprezentată prin două valori întregi (zi și lună) coincide cu data Crăciunului și FALSE în caz contrar.

15. Să se scrie o expresie logică (booleană) care să aibă valoarea true dacă un întreg, a, reprezintă un an bisect. Precizare : anii bisecți sunt multipli de 4, exceptând multiplii de 100, dar incluzând multiplii de 400.

CAPITOLUL 6

STRUCTURA GENERALĂ A UNUI PROGRAM PASCAL

Elementele ce urmează a fi descrise în capitolele următoare (operații de intrare/ieșire, instrucțiunile limbajului PASCAL, subprograme etc.) vor fi utilizate, desigur, pentru scrierea programelor. Înainte de a trece la abordarea acestora, considerăm că este util să prezentăm cititorului, familiarizat deja cu o serie de noțiuni mai simple, modul în care este organizat un astfel de program. Acest lucru îi va permite să înțeleagă mai bine modul în care se încadrează în construcția de ansamblu a unui program noțiunile ce vor fi prezentate în continuare.

Pornind de la un exemplu foarte simplu, cu ajutorul căruia vom identifica o parte din secțiunile caracteristice oricărui program PASCAL, vom prezenta apoi structura generală a unui astfel de program.

Exemplul 1. *Program pentru calculul ariei unui cerc.* Figura 6.1 conține listingul acestui program foarte simplu.

Dincolo de banalitatea programului, apar ca evidente cele două secțiuni principale ale unui program scris cu ajutorul limbajului PASCAL: secțiunea declarativă și secțiunea de instrucțiuni, prima având rolul de a grupa,

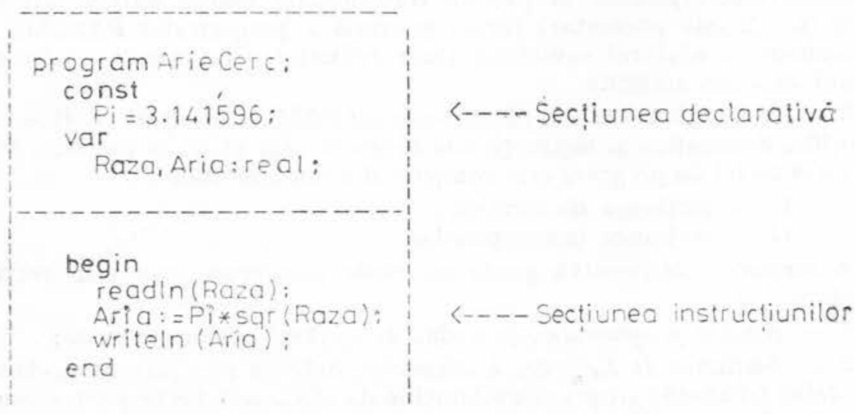


Fig. 6.1.

```
                     --------------------
                     | program ...        |  <-- Antetul programului
                     |--------------------|
                     | label ...          |  <-- Zona etichete
                     |                    |
                     |--------------------|
                     | const ...          |  <-- Definiţii constante
                     |                    |
                     |--------------------|
                     | type ...           |  <-- Declaraţii de tip
 Secţiunea           |                    |
 declarativă    -->  |--------------------|
                     | var ...            |  <-- Declaraţii de
                     |                    |      variabile
                     |--------------------|
                     | function ...       |
                     |  ---------------   |
                     | |               |  |  <--
                     | |               |  |     |
                     |  ---------------   |     |
                     |--------------------|     +--- Subprograme
                     | procedure ...      |     |
                     |  ---------------   |     |
                     | |               |  |  <--
                     | |               |  |
                     |  ---------------   |
                     |--------------------|
                     ----------------------
                     | begin              |
                     |   ...              |
                     |                    |
                    ==                   ==
 Secţiunea           |                    |
 instrucţiunilor --> |                    |
                     |                    |
                     |   ...              |
                     | end.               |
                     ----------------------
```

Fig. 6.2.

în principal, definiţiile şi declaraţiile necesare, iar cea de-a doua reprezentând ansamblul instrucţiunilor ce produc transformări asupra datelor utilizate. În figura 6.2 este prezentată forma generală a programelor PASCAL, evidenţiindu-se cu ajutorul cuvintelor cheie delimitarea în „zone" a celor două secţiuni mai sus amintite.

Trebuie menţionat faptul că un program PASCAL complet conţine toate definiţiile, declaraţiile şi instrucţiunile necesare. Aşa cum am precizat, structura unui astfel de program este compusă din două secţiuni :

I — secţiunea declarativă ;
II — secţiunea instrucţiunilor.

● **Secţiunea declarativă** poate cuprinde maximum şase alte secţiuni, grupate astfel :

1 — *Antetul programului*, precedat de cuvântul cheie **program ;**

2 — *Secţiunea de declarare a etichetelor*, definită cu ajutorul cuvântului cheie **label** (etichetă) grupează declaraţiile de etichete folosite pentru controlarea strictă a salturilor necondiţionate generate în program de instrucţiunile **goto ;**

3 — *Secțiunea de declarare a constantelor*, definită cu ajutorul cuvântului cheie **const**. Gruparea declarațiilor de constante într-o singură secțiune permite găsirea cu ușurință a acestora în cazul în care se dorește modificarea valorii unora dintre ele ;

4 — *Secțiunea de declarare a tipurilor*, definită cu ajutorul cuvântului cheie **type**, grupează declarațiile de noi tipuri de variabile specifice utilizatorului (în afara celor patru tipuri standard amintite anterior : **boolean**, **integer**, **real** și **char**) ;

5 — *Secțiunea de declarare a variabilelor*, definită cu ajutorul cuvântului cheie **var**, grupează declarațiile de variabile. De notat că, în mod obligatoriu, toate varabilele ce apar în program se declară împreună cu tipul lor, altfel spus, nu pot fi utilizate în program variabile nedeclarate ;

6 — *Secțiunea de declarare a funcțiilor și procedurilor*, care grupează toate declarațiile de funcții și proceduri, precedate de cuvintele cheie **function**, respectiv **procedure**.

În cazul în care, de exemplu, în program nu se folosesc etichete, secțiunea 1 dispare. Observația este valabilă și pentru celelalte secțiuni.

● **Secțiunea instrucțiunilor** este delimitată de cuvintele cheie **begin** și **end** și cuprinde toate instrucțiunile programului.

Subprogramele unui program PASCAL (funcțiile și procedurile) au ele însele aceeași structură cu cea prezentată în figura 6.2. Declararea fiecărui subprogram este precedată de cuvîntul cheie corespunzător (**function** sau **procedure**). Detalii vor fi furnizate în capitolele 10 și 11.

CAPITOLUL 7

OPERAȚII DE INTRARE/IEȘIRE

7.1. ELEMENTE GENERALE

Nu se poate concepe un program, care să se bucure de atributul de „generalitate", în absența așa numitelor „operații de intrare/ieșire" (citire/scriere). Exemplul de mai jos vine în sprijinul afirmației precedente. Să considerăm programul următor:

Exemplul 1.

```
program Jurin ;
{Calculul variației de nivel a unui lichid într-un vas capilar}
var Sigma, r, Ro, g, h : real ;
begin
      Sigma : = 0.0002 ;
      r : = 0.0005 ;
      g : = 9.81 ;
      Ro : = 1000.0 ;
      h : = (2*Sigma)/(r*Ro*g)
end.
```

Cea de-a cincea instrucțiune are un caracter general, însă, dacă dorim repetarea calculelor pentru o altă valoare de timp (alte valori Sigma, r, Ro, g), trebuie practic să rescriem primele patru instrucțiuni. Programul, scris astfel, este extrem de rigid.

În fapt, problema centrală a programării rezidă în a scrie programul o singură dată, însă în așa fel încât să poată fi utilizabil (fără modificări) pentru o gamă cât mai largă de seturi de date de intrare. Astfel, în „programul" de mai sus trebuie să asigurăm o modalitate de introducere a datelor (Sigma, r, Ro, g) la fiecare rulare a programului. Procedurile Read și ReadLn ne vor permite acest lucru. Programul de mai sus, pe lângă lipsa generalității, mai prezintă încă o deficiență : valoarea calculată pentru h rămâne pur și simplu invizibilă utilizatorului ! Pentru a rezolva acest tip de problemă, în PASCAL sunt prevăzute procedurile Write și WriteLn.

Într-o formă mai generală, programul poate fi rescris astfel :

```
    ...
    Read (Sigma) ;
    Read (r) ;
    Read (Ro) ;
    Read (g) ;
    h : = (2*Sigma)/(r*Ro*g) ;
    Write (h) ;
end.
```

sau, chiar mai concis, sub forma :

```
    ...
    Read (Sigma, r, Ro, g) ;
    h : = (2*Sigma)/(r*Ro*g) ;
    Write (h) ;
end.
```

ilustrându-se totodată faptul că procedura Read poate accepta mai mulţi parametri de intrare (în exemplul de mai sus : Sigma, r, Ro, g). Toţi aceşti parametri ai procedurii Read sunt **variabile** şi fiecăreia i se atribuie, prin citire, o unică valoare.

În mod similar, procedura Write admite la rândul său mai mulţi parametri, putându-se scrie, de exemplu :

Write (Sigma, r, Ro, g, h)

însă, spre deosebire de parametrii procedurii Read, aceştia pot fi **expresii**, nu doar simple variabile, tipărindu-se valorile acestor expresii.

Observaţie : din punctul de vedere al intrărilor şi ieşirilor există două modalităţi de procesare a datelor :

— modul de lucru **batch** în care datele de intrare sunt pregătite în avans (de exemplu : perforate pe cartele) iar rezultatele sunt tipărite la imprimantă, acestea fiind returnate utilizatorului după ce programul şi-a terminat execuţia ;

— modul de lucru **interactiv** în care datele de intrare sunt introduse de la terminalul de intrare în timpul execuţiei programului iar rezultatele sunt accesibile utilizatorului imediat după producerea lor (prin afişare la display sau listare la imprimantă) pe parcursul acestei execuţii.

7.2. REPREZENTAREA OPERAŢIILOR DE CITIRE/SCRIERE CU AJUTORUL FIŞIERELOR DE INTRARE/IEŞIRE

Operaţiile de intrare/ieşire nu sunt altceva decât transferuri de date între unităţile de intrare/ieşire ale sistemului de calcul şi memoria internă a calculatorului.



Fig. 7.1, a

Acceptând deocamdată definiţia „un **fişier** reprezintă o *colecţie organizată de date*", putem să ne reprezentăm aceste transferuri pe baza modelului din figura 7.1.

Precizare : Dacă ne imaginăm datele unui fişier amplasate într-o anumită ordine şi dacă operaţiile asupra datelor din fişier se efectuează în ordinea amplasării, atunci spunem că fişierul este **secvenţial.**

Limbajul PASCAL asociază identificatori **INPUT**, respectiv **OUTPUT** pentru fişierele standard de intrare, respectiv de ieşire (de obicei intrarea se efectuează de la tastatură iar pentru ieşirea standard se asociază fie afişajul catodic — display-ul — fie imprimanta).

Un fişier de intrare/ieşire este alcătuit din una sau mai multe linii, fiecare linie conţinând cel puţin un caracter. O linie este subdivizată **în câmpuri** şi se termină printr-un caracter special, numit **terminator de linie.**

Reprezentarea unui fişier de acest tip (numit şi **fişier text**) poate fi (fig. 7.1, *b*) :

174 293	▨	-50 22 29	▨	-39	▨

Fig. 7.1, b

în care s-au marcat prin haşurare terminatorii de linie, caractere în fapt invizibile operatorului. Precizăm că aceşti terminatori de linie sunt introduşi în fişierul de intrare la apăsarea tastei RETURN (CR + LF). Dincolo de reprezentarea de mai sus (cu un anumit grad de abstractizare), reprezentare pe care o vom folosi în cele ce urmează, putem să ne imaginăm şi reprezentări mai concrete pentru fişierele de intrare/ieşire, după cum urmează :

— în cazul perforării pe cartele, ca în fig. 7.2 ;

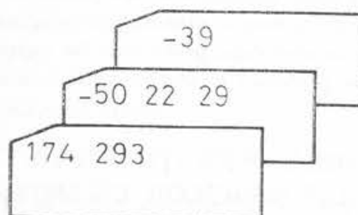


Fig. 7.2.

— la afişarea pe display sau la tipărirea pe hîrtie, ca în figrua 7.3.

```
174 293
-59 22 29
  -39
```

Fig. 7.3

Considerând prima dintre aceste trei reprezentări, putem discuta în detaliu despre procedurile standard de I/O (intrare/ieşire) ale limbajului PASCAL.

7.3. PROCEDURILE READ, READLN, WRITE, WRITELN

Prin **procedură** înţelegem un *program scris într-o formă specială astfel încât să poată fi apelat dintr-un alt program, acesta din urmă furnizând procedurii un set de parametri de intrare.* Despre proceduri vom discuta în detaliu în cadrul capitolului 10.

7.3.1. Procedura Read

● **Procedura Read** are ca formă generală :

read ([f,] var1[, var2, . . ., varN]) ;

în care s-au inclus între paranteze drepte parametrii opţionali ai procedurii.

Se observă că numărul de parametri de apel pentru această procedură nu este fix.

Prin f s-a notat numele fişierului de intrare. În cazul în care nu este precizat la apel, se consideră implicit fişierul standard de intrare (INPUT) (maniera de lucru cea mai utilizată) ; altfel, f poate fi numele unui fişier de pe disc, bandă etc. (Despre fişiere vom discuta în detaliu în capitolul 12).

În lista variabilelor var1, . . ., varN un singur parametru este obligatoriu (var1), ceilalţi sunt opţionali.

Exemple de apel

```
read (a, b, c) ;
read (k) ;
read (H, M, S).
```

Procedura Read tratează fişierul de intrare ca pe o succesiune de câmpuri (având mai puţină importanţă organizarea în linii). La un apel al procedurii Read se citesc unul sau mai multe câmpuri (N), avansându-se în fişier până la începutul câmpului următor. Câmpurile sunt separate prin blancuri, numărul acestora (>= 1) nefiind relevant.

Observaţie : La citirea datelor de tip char, blancul este considerat caracter, nu separator de câmp.

Există două *condiţii pentru funcţionarea corectă a procedurii Read :*

1. Numărul de câmpuri din fişierul de intrare trebuie să depăşească sau să fie egal cu numărul variabilelor precizate în apelul de procedură ;
2. Dincolo de cel de-al N-lea câmp din fişierul de intrare trebuie să existe un terminator de linie.

De reţinut că variabilele precizate în apelul procedurii primesc valori în ordinea scrierii, prin citirea secvenţială a câmpurilor.

Exemplul 2. Presupunând că fişierul de intrare este cel din fig. 7.4, a,

-14 222	▨	13 2 7	▨

↑

Fig. 7.4, a

la execuţia instrucţiunii Read (i, j, k) se vor face următoarele atribuiri:
i primeşte valoarea —14
j „ 222
k „ 13
următoarele valori din fişier (2 şi 7) rămânând disponibile pentru citirile ulterioare, avansul în fişier fiind efectuat ca în fig. 7.4, b.



Fig. 7.4, b

Exemplul 3. Pentru a înţelege mai bine cum lucrează procedura Read, să considerăm că ne aflăm în faţa tastaturii şi că rulăm un program care a ajuns la instrucţiunea Read (a, b, c, d), unde presupunem a, b, c, d de tip întreg. Dacă tastăm
12 39 ⟨RETURN⟩
constatăm că „nu se întâmplă nimic", adică programul nu-şi continuă execuţia, rămânând tot la linia (instrucţiunea) de mai sus. Într-adevăr, condiţia 1, din cele două precizate anterior, nu este încă îndeplinită. Tastând, în continuare
7 8 22 —555
observăm acelaşi efect. Pentru a fi îndeplinită condiţia 2, trebuie neapărat să apăsăm tasta ⟨RETURN⟩.
La apăsarea tastei ⟨RETURN⟩ programul îşi continuă execuţia, cu valorile 12 pentru a, 39 pentru b, 7 pentru c, şi 8 pentru d. Valorile 22 şi —555 vor putea fi citite la instrucţiunea Read următoare.

7.3.2. Procedura ReadLn

Procedura ReadLn are forma generală de apel:
readln [(f, var1, ...varN)];
adică similară procedurii Read, existând însă două **diferenţe majore**:

1. În primul rând, procedura ReadLn „priveşte" fişierul de intrare mai degrabă ca o succesiune de linii (fără a ignora organizarea acestora în câmpuri). La un apel al procedurii ReadLn se citesc 0, 1 sau mai multe (N) câmpuri din fişier, avansându-se apoi până la începutul liniei următoare. În acest mod anumite date din fişierul de intrare pot fi pierdute (ignorate).

Exemplul 4. Considerând fişierul de intrare reprezentat în fig. 7.5, a,

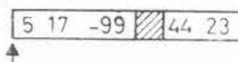


Fig. 7.5, a

după execuţia instrucţiunii readln (k, l) unde k, l sunt presupuse, pentru simplitate, de tip întreg avem situaţia:
k a primit valoarea 5
l „ 17
iar avansul în fişier este ilustrat în fig. 7.5, b,

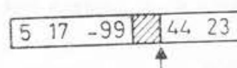


Fig. 7.5, b

valoarea —99 fiind irecuperabil pierdută.

2. A doua diferență constă în faptul că procedura ReadLn poate fi apelată fără parametri, sub forma

readln ;

efectul fiind de avans la linia următoare.

Observație : Această modalitate de utilizare este folositoare pentru „suspendarea execuției" unui program (de exemplu, când dorim să avem timpul necesar inspectării unui tabel de rezultate afișat pe ecran), continuarea făcându-se abia după apăsarea tastei ⟨RETURN⟩ (precedată eventual de un număr nedefinit de caractere, ignorate de către program).

Celelalte precizări făcute la procedura Read (de exemplu condițiile de funcționare corectă) rămân și aici valabile. Trebuie spus însă, că, pentru a ajuta operatorul în munca sa de introducere de date, procedurile Read și ReadLn trebuie precedate la execuție de mesaje către operator, indicându-i acestuia ce date are de introdus. Se va vedea în continuare cum se transmit aceste mesaje.

7.3.3. Procedurile Write și WriteLn

- Forma generală pentru **procedura Write** este :

write ([f,] par1[, par2, ...parN]) ;

La un apel al procedurii Write se scriu unul sau mai multe câmpuri consecutive, avansându-se în fișier la începutul câmpului următor.

- **Procedura WriteLn**, având forma generală :

writeln [(f, par1, par2, ...parN)] ;

lucrează la fel ca procedura Write, efectuând suplimentar un avans la linia următoare.

Exemplele de mai jos ilustrează modul de execuție al celor două proceduri.

Exemplul 5. Pentru X având valoarea 26 la instrucțiunea write (X) ;
se tipărește 26— —
Prin simbolul '—', urmând celor două cifre de mai sus, se indică poziția în care se va efectua (eventual) următoarea scriere.

Exemplul 6. PentruX având valoarea 45 la instrucțiunea writeln (X) ;
se tipărește 45 și cursorul trece pe linia următoare.

Exemplul 7. Pentru a, b, c având respectiv valorile 19, 53, —8, instrucțiunea
write (a, b, c) ;
produce rezultatul 1953 —8
iar instrucțiunea
write (a, ' ', b, ' ', c) ;
va tipări valorile distanțat :
19 53 —8

Exemplul 8. Alte variante de utilizare a instrucțiunilor Write și WriteLn:

```
write (6);
write (3*sin(x) + sqr(x)) ;
writeln (29*3*b) ;
```

Exemplul 9. Presupunem x, real, cu valoarea 19.2, atunci instrucțiunea:

```
write ('x = ', x) ;
```

va produce textul de ieșire:

```
x = 1.9200000000E + 01
```

Toate tipurile de date standard (integer, real, boolean, char) pot fi scrise cu ajutorul procedurilor standard Write și WriteLn, în schimb la citire nu se acceptă decât date de tip integer, real sau char. Parametrii de apel ai tuturor procedurilor amintite pot diferi ca tip, de exemplu:

```
read (i, x, c) ;
unde i — integer
     x — real
     c — char,
```

rămânând în grija operatorului să introducă valori corespunzătoare.

Fiecare dintre parametrii notați „parametru" („par") în cadrul formei generale a procedurilor Write și WriteLn poate avea una dintre următoarele **forme de reprezentare :**

e
e : e1
e : e1 : e2

în care e, e1 și e2 sunt **expresii**, având semnificațiile :

e — valoarea ce urmează a fi scrisă și care poate fi de tip char, integer, real, boolean sau poate fi un șir de caractere ;

e1 — numită **dimensiunea minimă a câmpului,** reprezintă un element opțional de control al formatului tipăririi. Este un număr natural care arată numărul minim de caractere ce vor fi scrise. În general, valoarea „e" se scrie în acest câmp, lăsându-se spații la stânga ei, dacă numărul de poziții ocupate este mai mic decât dimensiunea „e1". Dacă însă „e1" are o valoare „prea mică" pentru reprezentarea lui „e", se alocă mai mult spațiu. Numerele reale trebuie scrise astfel încât să fie precedate de cel puțin un spațiu sau de semnul minus. Dacă „e1" nu este precizat, se folosește pentru scriere o valoare implicită a câmpului (dependentă de implementare), corespunzător tipului expresiei „e".

e2 — numită **dimensiunea părții fracționare,** reprezintă un element opțional de control al formatului tipăririi, folosit numai în cazul în care „e" este o expresie de tip Real. Este un număr natural care specifică numărul cifrelor zecimale ce vor fi scrise după punctul zecimal (se spune că numărul va fi scris în format cu „virgulă fixă"). Dacă nu se specifică acest element, valoarea va fi scrisă în format cu „virgulă mobilă" adică folosind notația științifică și atâtea poziții pentru partea fracționară cât este posibil.

Reprezentarea în „virgulă mobilă" se numește astfel deoarece numărul de cifre zecimale memorat în calculator este fix, mereu același, în timp ce poziția punctului zecimal este mobilă în raport cu cifrele memorate, fiind urmărită în mod automat de către calculator în vederea efectuării calculelor aritmetice. În cadrul acestei reprezentări se memorează numai cele mai semni-

ficative cifre zecimale ale numărului real iar poziția punctului zecimal se păstrează cu ajutorul **factorului de scală.** Factorii de scală negativi determină mutarea punctului zecimal la stânga, iar cei pozitivi, la dreapta. De exemplu, într-un calculator ce poate memora numere reale formate din 6 cifre zecimale :

0.77	se memorează ca 7.70000	cu factor de scală −1
7.7	se memorează ca 7.70000	cu factor de scală 0
0.0077	se memorează ca 7.70000	cu factor de scală −3
8.7654321	se memorează ca 8.76543	cu factor de scală 0
24630871.6	se memorează ca 2.46309	cu factor de scală +7

Precizia limitată a acestei reprezentări determină rotunjirea forțată a numerelor cu prea multe cifre zecimale semnificative, ceea ce poate conduce la rezultate neașteptate.

Observație : Dacă valoarea ce urmează a fi scrisă este de tip boolean, se va scrie practic unul dintre identificatorii standard True sau False.

Variabilele de tip char și șirurile vor fi scrise la ieșire fără a mai fi încadrate între apostrofuri.

Exemplul 10.

write (' Rezultatul adunării este :')

va afișa textul :

Rezultatul adunării este :

Exemplul 11. Să considerăm că variabila x, de tip real, are valoarea 1991.315 Vom scrie instrucțiuni de scriere în mai multe forme posibile și rezultatul acestora

Instrucțiune	Rezultatul scrierii
write ('x = ', x)	x = 1.9913150000E + 03
write ('x = ', x : 14)	x = 1.9913150E + 03
write ('x = ', x : 9)	x = 1.99E + 03
write ('x = ', x : 12 : 6)	x = 1991.315000
write ('x = ', x : 9 : 6)	x = 1991.315000

Observație : La ultima instrucțiune, deoarece valoarea (9) a câmpului total de reprezentare este mai mică decât valoarea necesară, se realizează scrierea pe atâtea poziții cât este nevoie, dar nu se mai lasă spații la stânga numărului scris.

Exemplul 12. Considerăm că variabilele de tip întreg k și j au valorile 493, respectiv —55. Folosim aceeași metodă :

Instrucțiune	Rezultat
write ('k = ', k : 7)	k = 493
write ('k = ', k : 2)	k = 493
write ('k = ', k)	k = 493
write ('k = ', k : 5, 'j = ', j : 5)	k = 493j = —55
write (k, j)	493—55
write (k : 2, j : 5)	493 —55

Exemplul 13. În cazul în care dorim să precizăm utilizatorului datele pe care trebuie să le introducă pentru a putea executa un anumit program, se poate folosi o succesiune de instrucțiuni Write/Read (cu variantele posibile), ca în exemplul următor :

```
write (' Introduceți x : ');
readln (x);
```

ceea ce va genera tipărirea pe ecran a mesajului cuprins între caracterele apostrof, urmând ca operatorul să introducă de la tastatură valoarea variabilei x, ce va fi apoi citită de către instrucţiunea ReadLn a programului.

În final, trebuie precizat că datele de intrare se citesc întotdeauna de la stânga la dreapta, în cadrul aceleiaşi linii şi de la o linie la alta, în cadrul unui fişier sau al unei sesiuni de introducere date de tip interactiv. Informaţiile deja citite nu pot fi citite încă o dată.

În mod asemănător, datele de ieşire se scriu de la stânga la dreapta, în cadrul unei linii şi de la o linie la alta, nefiind posibil ca ele să fie şterse, rescrise sau completate. Aceste restricţii, impuse de caracteristicile dispozitivelor uzuale de intrare/ieşire ale sistemelor de calcul, trebuie avute în vedere în momentul scrierii programelor.

Trebuie, de asemenea, făcută o completare referitoare la structura generală a unui program scris în PASCAL. Am menţionat într-un capitol anterior că programul este format din două mari secţiuni : secţiunea declarativă şi secţiunea instrucţiunilor. Diagrama de sintaxă completată a unui astfel de program este prezentată în figura 7.6.

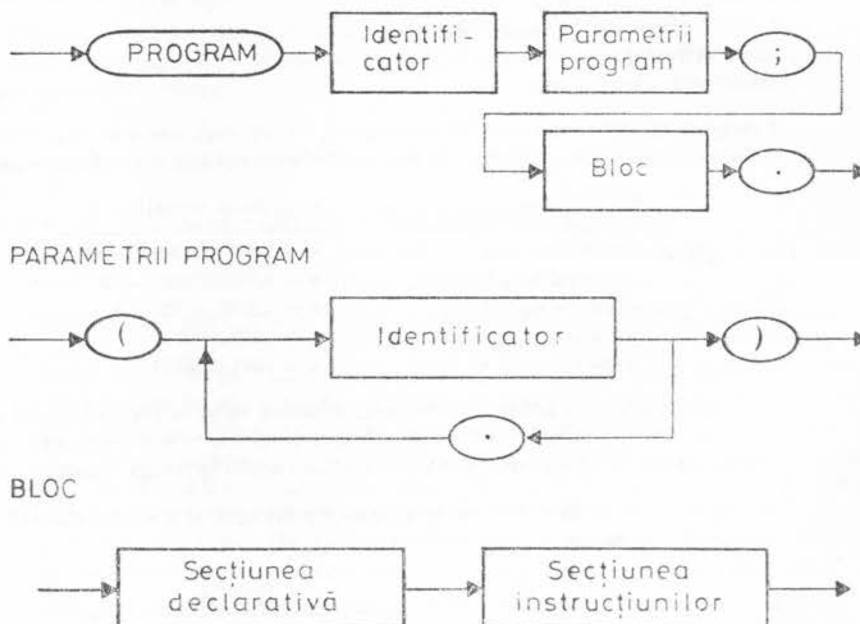


Fig. 7.6.

De exemplu,

program test (input, output)

va stabili un identificator („test") pentru programul creat şi va specifica faptul că acest program va realiza atât operaţii de citire, cât şi operaţii de scriere. „Input" şi „Output" sunt parametrii program, semnificaţia lor urmând a fi detaliată într-un capitol viitor. Am ţinut totuşi să menţionăm existenţa acestor parametri, deoarece în versiunile mai vechi ale limbajului un program nu poate realiza citiri dacă parametrul „Input" nu este specificat şi nu poate realiza scrieri dacă nu se specifică parametrul „Output".

● *Funcții standard utilizate împreună cu operațiile de intrare/ieșire :*

EOF — testează sfârșitul fișierului de date (End of File);
— ia valoarea True numai în cazul în care, prin citire, s-a trecut de ultimul „terminator de linie".

EOLn — testează sfârșitul unei linii de date (End of Line);
— ia valoarea True ori de câte ori s-a ajuns, prin citire, la un „termiminator de linie".

EXERCIȚII

1. Să se scrie un program PASCAL care citește trei valori întregi, pozitive, strict mai mici decât 1000 și le imprimă împreună cu media lor sub forma :

A = *** B = *** C = *** MEDIA = ***.**

2. Considerând R — raza unui cerc, LungCerc — lungimea cercului și Aria — suprafața acestuia (evident, variabile reale) scrieți secvența de instrucțiuni pentru tipărirea următoarelor informații :

Raza cercului este :	****.***
Lungimea cercului :	****.***
Aria cercului este :	*****.****

Am notat prin * un caracter destinat reprezentării valorilor variabilelor de mai sus.

3. De la tastatură se citește un număr exprimat în radiani. Să se convertească numărul în grade, minute, secunde și zecimi de secundă.

4. Scrieți un program care citește un număr și afișează sub formă de tabel numărul, pătratul său și numărul ridicat la puterea a treia.

5. O dată calendaristică introdusă de la tastatură sub forma a trei întregi (zi, lună, an) trebuie afișată pe ecran sub forma zi/luna,/an, păstrând din valoarea anului doar ultimele două cifre. Scrieți fragmentul de program care implementează această funcție.

6. Două intervale de timp sunt exprimate în ore, minute, secunde, zecimi și sutimi de secundă. Să se calculeze suma lor exprimată în același mod.

CAPITOLUL 8

INSTRUCŢIUNILE DE BAZĂ ALE LIMBAJULUI PASCAL

Pentru a genera rezultatele dorite, un program trebuie să acţioneze asupra datelor într-un mod bine precizat. Descrierea acestor acţiuni se face cu ajutorul instrucţiunilor limbajului de programare care pot fi simple (instrucţiunea de atribuire) sau structurate.

8.1. INSTRUCŢIUNEA DE ATRIBUIRE (DE ASIGNARE)

Diagrama de sintaxă a instrucţiunii de atribuire este redată în figura 8.1.

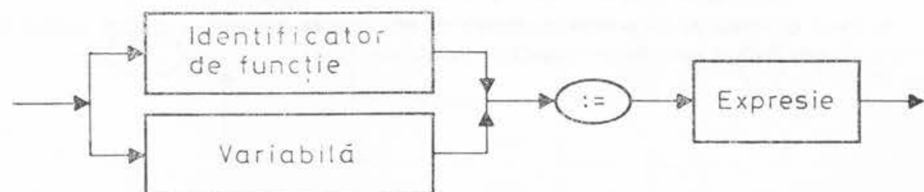


Fig. 8.1.

În urma executării acestei instrucţiuni, mărimea din stânga operatorului de atribuire „:=" primeşte valoarea rezultată prin evaluarea expresiei.

Mărimile atribuite pot fi de orice tip, cu excepţia tipului fişier despre care vom discuta într-un capitol viitor. Ca regulă generală, mărimea din membrul stâng şi expresia din membrul drept trebuie să fie de acelaşi tip. Se admite o singură excepţie : dacă partea stângă este de tip real şi partea dreaptă este de tip integer, înainte de atribuire se face automat conversia la tipul membrului stâng.

Observaţii :

1. În PASCAL, la declarare, variabilele nu pot primi valori iniţiale ;
2. Semnul de atribuire „:=" este diferit de operatorul relaţional de egalitate "=".

După cum s-a arătat în capitolul 5, o **expresie** poate fi o expresie *simplă* sau o *comparaţie.* Expresia reprezintă o succesiune de operaţii între diferite valori, având rolul de a calcula o nouă valoare pe baza celor deja existente.

Vom prezenta în continuare **diagrama de sintaxă a expresiilor simple,** sintaxa comparațiilor fiind prezentată anterior :

— expresie simplă (fig. 8.2) ;

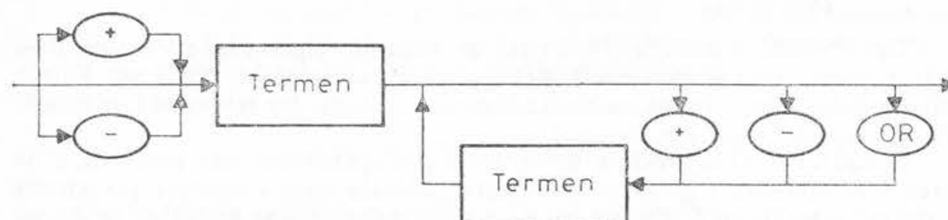


Fig. 8.2.

— termen (fig. 8.3) ;

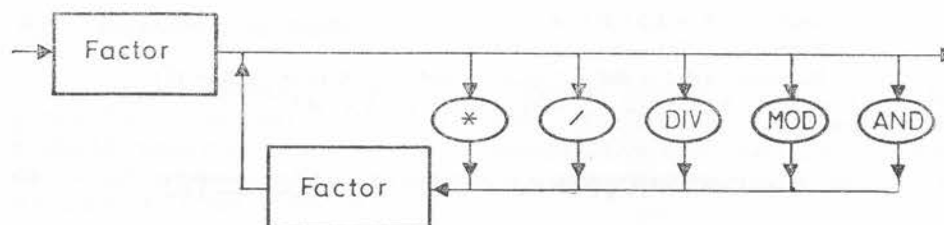


Fig. 8.3.

— factor (fig. 8.4).

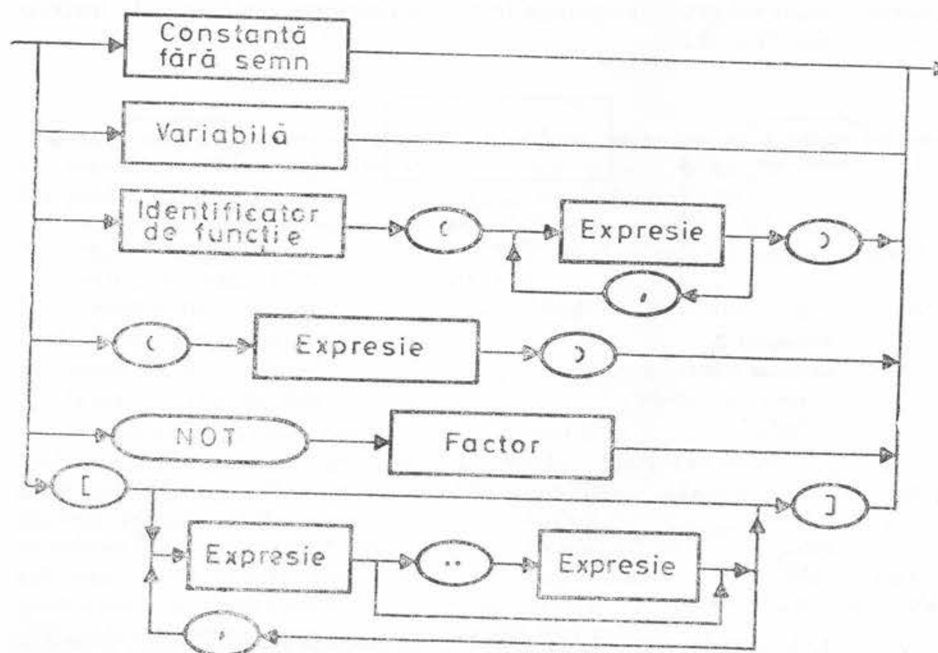


Fig. 8.4.

Exemple de expresii

3—(A mod B)
1/(sqrt (x*y + y*y))
(A + B) < C

Tipul valorii rezultate din calcul se numeşte tipul expresiei. Cea de-a treia expresie este o expresie logică (valoarea sa poate fi True sau False). Primele două expresii sunt expresii aritmetice, valorile lor reprezintă un număr întreg respectiv un număr real.

În cadrul unei expresii, ordinea execuţiei operaţiilor este esenţială şi de aceea s-au introdus regulile de prioritate generală între operatori, prezentate în cadrul capitolului 5. Operatorii cu aceeaşi prioritate se aplică de la stânga la dreapta. Priorităţile se pot modifica prin folosirea parantezelor, regula de desfacere a acestora fiind cea din aritmetica elementară.

Exemplul 1.

Expresii : X = 1.5 ; P < = Q ;
Expresii simple : —x ; a + b ; x*x— 1 ;
Termenii : a*b ; p **and** q ; (x < y) **and** (y < z); (x + y)/(x— y) ;
Factori : A ; 25 ; (x + y + z) ; sin(x) ; **not** achitat ;

8.2. INSTRUCŢIUNEA COMPUSĂ (SECVENŢA)

Instrucţiunea compusă este formată dintr-o listă de instrucţiuni separate prin caracterul „;" şi cuprinse între cuvintele cheie **begin** şi **end**, instrucţiuni care se execută una după alta, în ordinea apariţiei lor în listă. Precizăm că şi „corpul" unui program (secţiunea instucţiunilor) are tot forma de instrucţiune compusă (fig. 8.5).

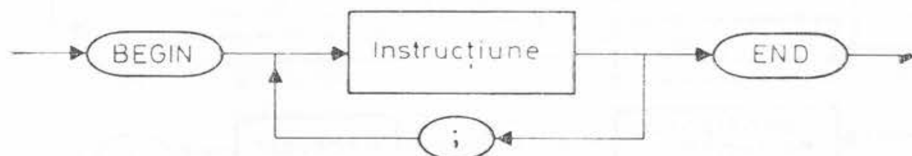


Fig. 8.5.

Exemplul 2.

```
program test ;
  var sum : integer ;
  begin
     sum : = 3 + 5 ;
     writeln(sum, -sum)
     writeln(sum, -sum)
  end.
```

Observaţii

1. În limbajul PASCAL, caracterul „;" se foloseşte pentru a separa instrucţiunile şi nu pentru a le încheia. Acest caracter nu face parte din instrucţiune. Dacă cineva ar scrie caracterul „;" după instrucţiunea „WriteLn(sum, -sum)" din

exemplul anterior, ar crea în acest mod o instrucţiune vidă, care nu implică nici o acţiune, plasată între „;“ şi simbolul **end**. Acest lucru este lipsit de importanţă în cazul aflat în discuţie, dar o plasare incorectă a caracterului „;“ poate genera uneori neplăceri.

2. După cum rezultă şi din diagrama de sintaxă, o instrucţiune compusă poate face parte din lista de instrucţiuni a unei alte instrucţiuni compuse, caz în care perechile, **begin**, **end** se grupează după principiul parantezelor.

8.3. INSTRUCŢIUNILE REPETITIVE

Instrucţiunile repetitive specifică faptul că anumite instrucţiuni se execută de mai multe ori. Dacă numărul de execuţii repetate este cunoscut, se foloseşte de obicei instrucţiunea **for**. În celelalte cazuri se pot folosi instrucţiunile **repeat** sau **while**.

8.3.1. Instrucţiunea While

Diagrama de sintaxă a acestei instrucţiuni este dată în figura 8.6. Expresia din componenţa acestei instrucţiuni trebuie să fie de tip logic şi este evaluată înainte de fiecare reluare a execuţiei. Din acest motiv, este de dorit ca această expresie să fie cât mai simplă pentru a nu consuma în mod repetat un timp de lucru substanţial.

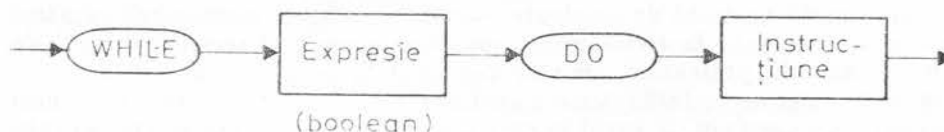


Fig. 8.6.

Instrucţiunea care apare după cuvântul cheie **do**, precizează operaţiile ce trebuie executate repetitiv, şi formează ceea ce se numeşte corpul ciclului. Ea poate fi o singură instrucţiune sau o instrucţiune compusă.

● *Succesiunea instrucţiunilor în cadrul instrucţiunii* **while** este :

1. Testarea condiţiei precizate prin „expresie“. Dacă aceasta nu este îndeplinită, se continuă cu pasul 4. ;
2. Execuţia instrucţiunii componente, ştiind că este îndeplinită condiţia ;
3. Revenire la pasul 1. ;
4. Ştiind că nu este îndeplinită condiţia, se continuă execuţia celorlalte instrucţiuni din program.

Datorită existenţei revenirii de la pasul 3 la pasul 1, structurile de control repetitive au mai fost numite şi „**bucle**“. Instrucţiunea componentă trebuie scrisă în aşa fel încât să modifice condiţia, astfel încât aceasta să primească valoarea False măcar în cadrul ultimei iteraţii. În cazul în care acest deziderat nu este realizat, pot să apară aşa-numitele „bucle infinite“ care se execută de un număr nedefinit de ori, împiedicând executarea normală, în continuare, a programului. Dacă de la început valoarea condiţiei este False, instrucţiunea componentă nu se execută nici măcar o dată.

Exemplul 3. Programul care citeşte o listă de numere, calculează şi afişează suma acestora.

• **Varianta 1 :**

```
program Sum1 ;
var
    suma, element, contor, dim : integer ;
begin
    suma : = 0 ;
    contor : = 0 ;
      {Urmează citirea dimensiunii listei}
    read (dim) ;
      while contor <dim do
        {citeşte următorul element şi îl adaugă sumei}
        begin
              read (element) ;
              suma : = suma + element ;
              contor := contor + 1
        end ;
    write (' suma numerelor întregi din listă este ', suma) ;
      writeln
end.
```

Datele de intrare trebuie introduse în următoarea formă :

6

249 100 404 139 335 157

în care 6 reprezintă dimensiunea listei, iar cea de-a doua linie conţine efectiv elementele acesteia.

În această variantă de rezolvare, variabila „contor" memorează numărul de execuţii repetate în ciclul **while**, indicând momentul terminării listei de numere. Metoda prezentată nu este convenabilă în cazul unor liste foarte lungi şi de aceea se preferă folosirea unei alte metode de detectare a sfîrşitului unei liste. De exemplu, în cazul în care ştim că în listă nu pot să apară decât numere pozitive, putem marca sfârşitul acesteia înscriind în listă, pe ultima poziţie, un număr negativ. În acest caz, datele de intrare se pot scrie astfel :
249 100 404 139 335 157 —1
iar programul care le prelucrează poate avea o formă mai simplă :

• **Varianta 2 :**

```
program sum2 ;
var
    suma, element : integer ;
begin
    suma : = 0 ;
    {citeşte primul număr întreg al listei}
    read (element) ;
    while element > = 0 do
          {adaugă un element în sumă, citeşte următorul element}
          begin
                suma : = suma + element ;
                read (element)
          end ;
    write ('suma numerelor întregi citite este ', suma) ;
    writeln
end.
```

In această variantă de program, variabila „element" are seminificaţii diferite la momente de timp diferite. În mod normal, ea conţine valoarea elementului ce se adaugă sumei, dar la sfârşitul listei conţine un terminator care nu se adaugă sumei. De asemenea, se presupune că în listă există cel puţin un element, altfel programul nu se poate executa.

În general, este greu să se stabilească un element cu rol de terminator, care să nu poată fi confundat cu datele. Limbajul PASCAL furnizează o modalitate de detectare a sfârşitului listei intrărilor, care să nu depindă de datele propriu-zise, aceasta fiind funcţia standard EOF (End Of File), care are valoarea True dacă şi numai dacă toate datele de intrare au fost citite.

Observaţie : Valoarea funcţiei EOF se testează şi furnizează informaţii utile, numai în cazul în care cea mai recentă operaţie a fost ReadLn şi nu Read.

- **Varianta 3 :**

Pentru buna funcţionare a programului, datele de intrare trebuie introduse în forma :

```
249
100
404
139
335
157
program sum3;
var
     suma, element: integer;
     begin
     suma:=0;
     while not EOF do
          {citeşte următorul element şi îl adaugă sumei}
          begin
               read (element) ;
               suma:=suma+element;
               readln
          end ;
     write ('suma numerelor întregi citite este ', suma) ;
     writeln
end.
```

În cazul în care datele sunt introduse de la tastatură, sfârşitul de fişier se marchează prin apăsarea unei anumite combinaţii de taste, cum ar fi, de exemplu, ⟨CTRL⟩ ⟨Z⟩ + ⟨RETURN⟩.

Exemplul 4. Calculul puterii n (n, număr natural) a numărului real x.

```
program Putere ;
    var x, y : real ;
        contor, n : integer ;
  begin
     read(x);
     readln(n);
     y:=1;
     contor:=n;
    while contor > 0 do
       begin
         y:=y*x;
         contor:=contor — 1
```

```
        end ;
    writeln(y) {y reprezintă puterea n a lui x}
end.
```

Aceste variante de rezolvare a exemplului propus, reprezintă tipuri de soluţii întâlnite în majoritatea programelor PASCAL.

8.3.2. Instrucţiunea Repeat

Am precizat mai înainte că, în cazul folosirii instrucţiunii **while**, este posibil ca numărul de execuţii repetate să fie zero (în cazul în care condiţia testată la începutul ciclului are valoarea iniţială False). Există totuşi situaţii în care dorim ca execuţia corpului ciclului să se efectueze cel puţin o dată. Acest lucru se realizează în PASCAL cu ajutorul instrucţiunii **repeat**, care are diagrama de sintaxă prezentată în figura 8.7

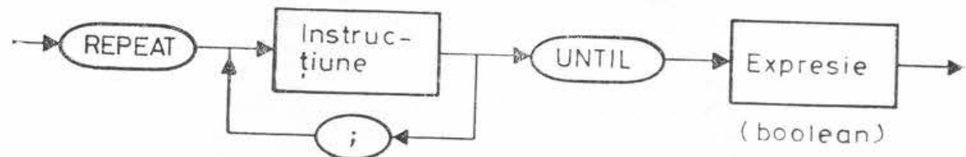


Fig. 8.7.

În interiorul buclei pot exista mai multe instrucţiuni, fără a fi grupate cu ajutorul cuvintelor cheie **begin**, **end**, deoarece cuvintele cheie **repeat** şi **until** îndeplinesc şi acest rol.

● *Succesiunea operaţiilor* este :

1. Executarea instrucţiunii (instrucţiunilor) cuprinse între **repeat** şi **until** ;
2. Testarea condiţiei. Dacă aceasta are valoarea False, se reia de la pasul 1. ;
3. Ştiind că valoarea condiţiei este True, se execută restul programului.

Instrucţiunea **repeat** se mai numeşte şi **ciclu (buclă) cu test final**, în timp ce instrucţiunea **while** poartă numele de **ciclu (buclă) cu test iniţial**. Instrucţiunea **repeat** se încheie în momentul în care condiţia devine adevărată (True), în timp ce instrucţiunea **while** se încheie în momentul în care condiţia nu mai e îndeplinită (False).

Exemplul 5. Cunoscând numărul natural n, să se scrie valoarea sumei $s(n) = 1 + 1/2 + 1/3 + ... + 1/n$

Rezolvare :

```
program suma1;
var
    n: integer;
    s: real;
begin
    read(n);
    writeln(n);
  s:=0;
  repeat
       s:=s + 1/n;
       n:=n—1
  until n = 0;
  writeln(s)
end.
```

8.3.3. Instrucțiunea For

Ciclurile (buclele) al căror *număr de iterații este cunoscut* se mai numesc și **cicluri (bucle) cu contor.** Ele sunt folosite destul de frecvent și au drept trăsătură caracteristică utilizarea unei variabile de control numită contor care ține evidența numărului de iterații. Contorul primește la începutul ciclului o valoare inițială ; în interiorul ciclului este incrementat (sau decrementat) ; ciclul se încheie în momentul în care contorul atinge valoarea finală prestabilită. În limbajul PASCAL, implementarea acestui tip de ciclu este realizată cu ajutorul instrucțiunii **for**, care are diagrama de sintaxă din figura 8.8.

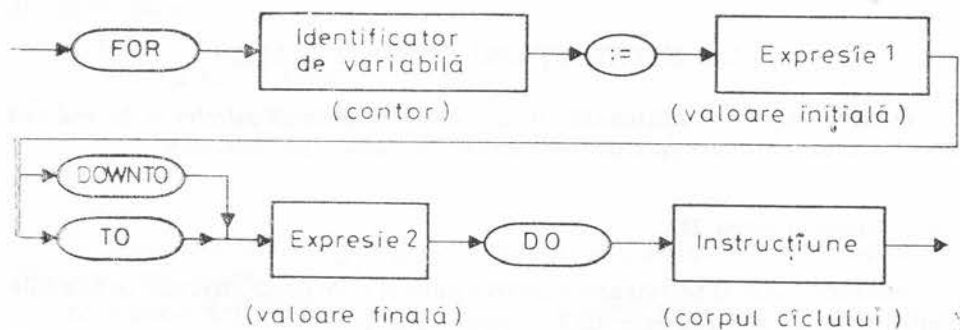


Fig. 8.8.

Corpul ciclului se execută de N ori, unde N se evaluează o singură dată, la început, după formula :

N = expresie 2 − expresie 1 + 1 (în cazul folosirii simbolului **to**)

N = expresie 1 − expresie 2 + 1 (în cazul folosirii simbolului **downto**)

Contorul trebuie să fie de tip scalar, cu excepția tipului real și trebuie declarat la fel ca orice altă variabilă. Expresiile (1 și 2) trebuie să fie compatibile din punctul de vedere al atribuirii cu tipul contorului.

Dacă numărul iterațiilor evaluat, N, este negativ, corpul ciclului nu se execută nici o dată.

Contorul se inițializează cu valoarea „expresiei" și se incrementează (forma **to**) sau se decrementează (forma **downto**) cu 1, în mod automat, după fiecare iterație, până ajunge la valoarea „expresie2".

Nu este permis ca instrucțiunile ce formează corpul ciclului să modifice valoarea contorului (printr-o atribuire, de exemplu). Ele însă o pot folosi. Modificarea contorului nu este semnalată ca eroare de compilare, în schimb produce erori de execuție.

Exemplul 6. Cunoscând numărul natural n, să se scrie valoarea sumei s(n) = = 1 + 1/2 + 1/3 +...+ 1/n

Rezolvare :

```
program suma2;
var
    i, n : integer;
    s : real ;
```

```
begin
      read(n) ;
      writeln(n) ;
      s : = 0 ;
      for i : = n downto 1 do s : = s + 1/i ;
      writeln(s)
end.
```

În acest exemplu, corpul ciclului este format dintr-o singură instrucţiune. Precizăm că, la fel ca şi în cazul instrucţiunii **while**, corpul ciclului poate fi o instrucţiune compusă.

8.4. INSTRUCŢIUNI CONDIŢIONALE

O **instrucţiune condiţională** (**if** sau **case**) realizează *selectarea în vederea execuţiei a unei singure instrucţiuni, dintre mai multe posibile.*

8.4.1. Instrucţiunea If

Instrucţiunea **if** selectează o instrucţiune dintre două alternative posibile Diagrama sa de sintaxă este dată în figura 9.8, a.

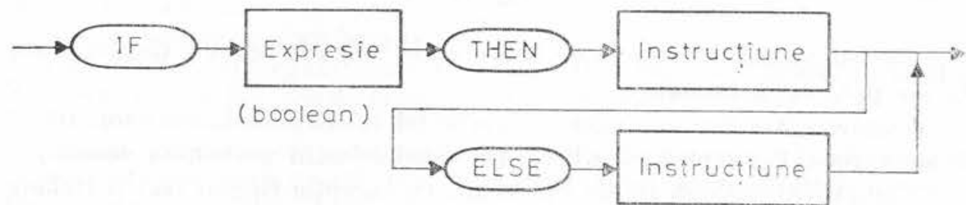


Fig. 8.9, a

Expresia reprezintă condiţia ce trebuie testată şi trebuie să fie de tip logic.

Succesiunea operaţiilor este următoarea :

1. Testarea condiţiei ;
2. Dacă valoarea condiţiei este True, se execută instrucţiunea care apare după cuvântul cheie **then** şi apoi se continuă cu pasul 4. ;
3. Dacă valoarea condiţiei este False, se execută instrucţiunea care apare după cuvântul cheie **else** şi apoi se continuă cu pasul 4. ;
4. Se continuă cu excepţia următoarelor instrucţiuni ale programului.

În cazul în care ramura din figura 8.9, b



Fig. 8.9, b

nu apare, succesiunea operațiilor este :

1. Se testează condiția ;
2. Dacă valoarea condiției este True, se execută instrucțiunea care apare după cuvântul cheie **then** ;
3. Se execută următoarele instrucțiuni ale programului.

„Instrucțiune" poate fi o singură instrucțiune sau o instrucțiune compusă.

Ambiguitatea sintactică ce poate să apară într-o formulare de tipul :

if expr1 **then** **if** expr2 **then** instrucțiune1 **else** instrucțiune2

se rezolvă în cadrul limbajului PASCAL aplicând regula : „**else** *corespunde celui mai apropiat* **if** *(precedent)*". Deci, putem scrie, echivalent :

```
if expr1 then
    begin
        if expr2 then
            instrucțiune1
        else
            instrucțiune2
    end
```

În toate cazurile care pot genera ambiguități, se recomandă folosirea cuvintelor cheie **begin** și **end**.

Exemplele date în continuare ilustrează *modul de utilizare a instrucțiunii* **if-else**.

Exemplul 7. Determinarea maximului dintre trei numere X, Y, Z.

• *Varianta 1.*

```
if (X > Y) and (X > Z) then
    Max:=X
else
    if Y > Z then
        Max : = Y
    else
        Max:=Z;
```

• *Varanta 2.*

```
if X > = Y then
    if X > = Z then
        Max:=X
    else
        Max:=Z
else
    if Y > = Z then
        Max:=Y
    else
        Max:=Z;
```

Exemplul 8. Considerăm că de pe mediul de intrare se citește o secvență de nu mere pozitive. Sfârșitul acestei secvențe este marcat printr-un număr negativ. Pentru fiecare număr strict pozitiv se va afișa pe ecran o linie de steluțe, numărul acestora fiind egal cu valoarea elementului respectiv din secvență. Pentru un element egal cu zero, la display se va tipări doar caracterul '0'. Presupunem că valoarea numerelor este mai mică decât 80.

```
program Histograma ;
    var star, element : integer ;
    begin
        read (element) ;
        while element > = 0 do
            begin
                if element = 0 then
                    write(element:1)
            else
                    begin
                        star:=0 ;
                        while star < element do
                            begin
                                write('*');
                                star:=star+1
                            end
                    end;
                writeln;
                read (element)
            end
    end.
```

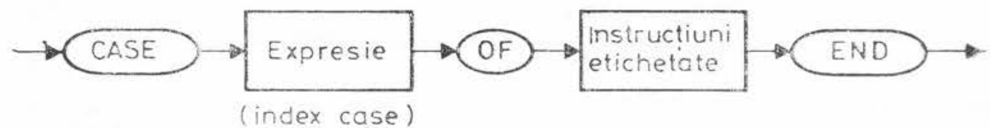
De exemplu, în cazul secvenței 5, 3, 0, 4, −2, pe displai se va afișa:

```
*****
***
0
****
```

8.4.2. Instrucțiunea Case

Această instrucțiune *realizează selectarea în vederea execuției a unei singure instrucțiuni dintre mai multe posibile.*

INSTRUCȚIUNEA CASE



INSTRUCȚIUNI ETICHETATE

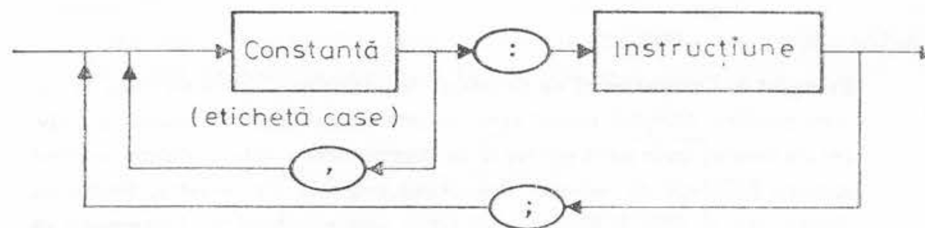


Fig. 8.10.

Expresia se mai numeşte şi *indexul instrucţiunii* **case**, iar constantele ce pot să apară se numesc *etichete* şi reprezintă valorile posibile ale indexului. Efectul instrucţiunii **case** este următorul : se evaluează indexul şi apoi se execută o singură instrucţiune a corpului selecţiei şi anume instrucţiunea ce are ca etichetă valoarea indexului.

Tipul indexului trebuie să fie scalar, mai puţin tipul real, iar etichetele trebuie să fie constante distincte de acelaşi tip cu idnexul.

Înaintea unei instrucţiuni pot fi puse mai multe etichete, separate prin virgulă ; această alternativă se foloseşte atunci cînd dorim ca programul să execute o aceeaşi operaţie pentru mai multe valori distincte ale indexului.

Etichetele instrucţiunii **case** nu reprezintă etichete obşnuite ale programului şi nu pot fi referite de către o instrucţiune **goto** (v. capitolul 14). Ele pot fi scrise într-o ordine arbitrară dar trebuie să fie unice în cadrul instrucţiunii **case**.

Dacă valoarea indexului nu coincide cu nici o etichetă, se va selecta instrucţiunea vidă. Anumite implementări ale limbajului acceptă cuvîntul cheie **else** (sau **otherwise**) pentru a prefixa instrucţiunea ce se execută în situaţia în care indexul nu coincide cu nici una dintre etichetele structurii **case** (exemplul 10).

Exemplul 9. Presupunând că i este o variabilă de tip întreg, iar x şi y variabile de tip real, se poate scrie :

```
case i of
        0: x:=0 ;
        1: x:=sin (y) ;
        2: x:=cos (y);
        3: x:=exp(y) ;
        4: x:=ln(y)
end
```

Exemplul 10. Considerând variabila de tip caracter Tasta se poate scrie următorul program, conţinând structura **case**

```
program TipTasta ;
var Tasta: char;
  begin
  Tasta:='0';
while Tasta < > '!' do
  begin
  readln (Tasta);
  case Tasta of
     'a','e','i','o','u'                    :writeln('Vocala');
     'o'..'9'                               :writeln('Cifra');
     'b'..'d','f'..'h','j'..'n','p'..'t',
     'v'..'z'                               :writeln('Consoana') ;
     ' '                                    :writeln('Blank');
  else                                      :writeln('Alt caracter')
  end
  end
end.
```

Aşa cum se observă în exemplul de mai sus, dacă pentru un grup de valori consecutive ale etichetelor se doreşte execuţia aceleiaşi instrucţiuni, se poate folosi forma de etichetare :

ValMin..ValMax : Instrucţiune

Este permisă şi construirea de variante mixte cum ar fi :

ValMin..ValMax, Val1, Val2 : Instrucţiune

Observaţie : Pe lângă instrucţiunile prezentate în cadrul acestui capitol, limbajul PASCAL utilizează şi instrucţiunile **with**, respectiv **goto**, a căror prezentare va fi făcută în cadrul altor capitole (capitolul 12, capitolul 14). Aceste instrucţiuni folosesc elemente a căror prezentare nu a fost încă făcută.

EXERCIŢII

1. Se consideră declaraţiile de variabile :

```
var
    a, b, c:integer ;
    x, y, z:real ;
    p:char ;
    t, f:boolean ;
```

Care dintre următoarele atribuiri sunt incorecte şi de ce ?

a. a: = x—trunc(x) ;
b. y:=b/c ;
c. z:=a **mod** c ;
d. f:=t **and** (ord(p)='p');

2. Se consideră declaraţiile de variabile :

```
var
    x, y:real ;
    m, n:integer ;
    ck:char ;
    b, b1:boolean ;
```

Care dintre următoarele atribuiri sunt incorecte ?

a. y:=x+y ;
b. x:=m*y;
c. m:=x*y ;
d. y:=m*x+b ;
e. y:=y+13.6 ;
f. ck:=m+n;
g. b:=false ;
h. b1:=b ;
i. x:=y+ck ;
j. b1:=x < y ;

3. Presupunând că sunt declarate următoarele variabile :

```
var
    i:integer ;
    j:1..10 ;
    c:char ;
    s:(masculin, feminin) ;
```

Să se precizeze care dintre următoarele instrucţiuni sunt corecte :

a. i:=1 ;
b. j:=i ;
c. c:='i' ;
d. c:=i ;
e. c:=chr(i) ;
f. i:=ord(c) ;
g. s:=i ;
h. s:=succ(s);

4. Cum se execută o instrucţiune **for** ?

5. Echivalaţi instrucţiunea :
for i:=MaxInt **downto** MaxInt—100 **do** writeln(i:6) ;
cu ajutorul instrucţiunii **while**.

6. Calculaţi suma $S = \sum_{k=1}^{n} (k^2 - k)$ cu ajutorul instrucţiunii **for**.

7. Ce eroare conţine următoarea instrucţiune ?

```
for j:=i to 12 do
   if j >= 6 then j:=j+1
   else j:=j—1 ;
```

8. Ce afişează următoarea secvenţă de program ?

```
Pı odus:=1 ;
Contor:=2;
while Contor <=5 do
Produs:=Produs*Contor ;
Contor:=Contor+1 ;
writeln(Produs) ;
```

9. Scrieţi un segment de program care calculează suma :
1+2+3+...+(N—1)+N
şi verifică dacă rezultatul este N*(N+1)/2.

10. Folosind instrucţiunea **repeat**, calculaţi suma $S = \sum_{k=1}^{n} \frac{1}{k^2}$.

11. Evidenţiaţi asemănările şi deosebirile dintre instrucţiunile repetitive **while**, **for** şi **repeat**.

12. Ce valoare va fi atribuită variabilei F în cadrul următoarei instrucţiuni **if**, dacă variabila viteză are valoarea 75 ?

```
if Viteza > 35 then
   F:=20.0
else if Viteza > 50 then
   F:=40.0
else if Viteza > 75 then
   F:=70.0;
```

13. Să se scrie un program ce afişează rădăcinile unei ecuaţii de gradul doi, considerându-se ca date de intrare coeficienţii reali a, b, c.

14. Să se scrie un program care, primind un unghi X şi o precizie EPS (>0), calculează valoarea funcţiei SIN(X) cu o precizie (relativă) EPS, folosind dezvoltarea în serie :

$$SIN(X) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{5^1}{5!} - \ldots, \text{ cu x număr real.}$$

15. Scrieţi un fragment de program care citeşte trei întregi reprezentând data curentă (zi, luna, an) urmaţi de trei întregi reprezentând ziua de naştere a unei persoane şi apoi calculează şi afişează vârsta persoanei în număr de ani.

16. De la tastatură se introduce o listă de numere întregi pozitive. Se cere să se afişeze valoarea maximă depistată în listă. Dimensiunea listei este precizată înainte de a fi introduse elementele ce o compun.

17. Cunoscându-se N, întreg pozitiv introdus de la tastatură, să se calculeze şi să se afişeze suma :

$$S = \sum_{k=1}^{n} (-1)k * k!$$

18. De la tastatură se introduce o dată calendaristică sub forma a trei întregi (zi, luna, an). Se cere să se afişeze data sub forma „zi—luna—an“, în care luna să apară cu numele ei şi nu ca număr întreg.

19. Scrieţi un program pentru calcularea rădăcinii pătrate a unui număr real pozitiv x, utilizând metoda lui Newton, bazată pe şirul convergent :

$$a_n = \frac{1}{2}\left(a_{n-1} + \frac{x}{a_{n-1}}\right); \ a_0 = 1.$$

CAPITOLUL 9

TIPURI DEFINITE ÎN PROGRAM (TIPURI DE DATE UTILIZATOR)

După cum s-a arătat în capitolul 5, un **tip** reprezintă o *mulțime de valori căreia i se poate atașa un nume.* Numărul elementelor acestei mulțimi se numește **cardinalitatea tipului.**

În limbajul PASCAL programatorul are posibilitatea de a folosi **tipurile standard** (integer, boolean, char, real) sau de a defini el însuși tipuri care reprezintă uneori **structuri de date complexe.**

Componentele unor astfel de structuri se reduc în ultimă instanță la „atomi" descriși prin tipurile simple ale limbajului : tipurile scalare și tipul pointer. Programatorul dispune și de posibilitatea de a defini *tipuri scalare proprii.* Mulțimea valorilor oricărui tip scalar este o *mulțime ordonată.* Aceasta este o proprietate specifică tipurilor scalare.

● În continuare sunt prezentate **diagramele de sintaxă pentru definirea tipurilor utilizator.**

— definiție de tipuri (fig. 9.1) ;

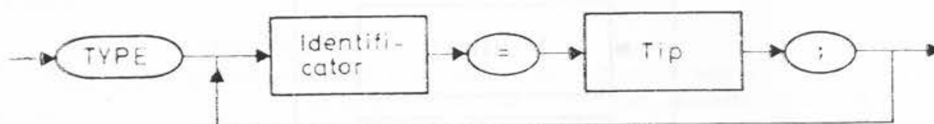


Fig. 9.1.

— tip (fig. 9.2) ;

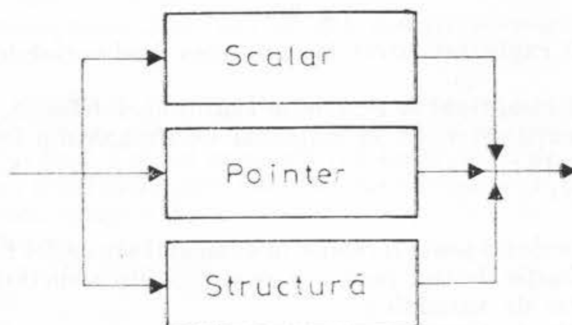


Fig. 9.2.

— scalar (fig. 9.3) ;

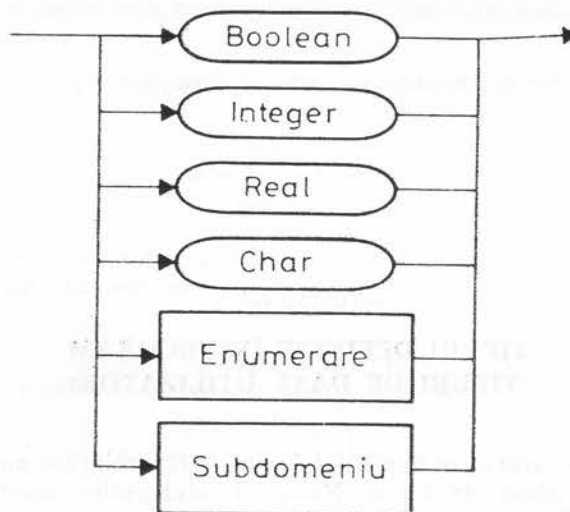


Fig. 9.3.

— structură (fig. 9.4).

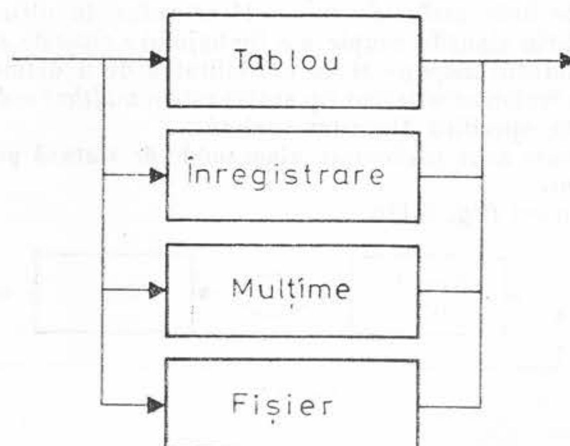


Fig. 9.4.

Tipul poate fi explicitat direct la declararea unei variabile :

var x : **tip** ;

În alte cazuri însă, tipul se descrie în cadrul unei definiții, atașându-i-se totodată și un identificator, ca în diagrama de sintaxă din figura 9.1. Va apare astfel secvența :

type numetip=tip ;

var x : numetip ;

Un *identificator de tip* poate fi referit în domeniul său astfel :

— într-o altă definiție de tip, pentru a se construi o structură complexă ;

— într-o declarație de variabile ;

— într-o declarație de funcție ;

— într-o listă de parametri formali.

Tipul pointer este folosit pentru descrierea și prelucrarea structurilor de date dinamice și va fi prezentat în capitolul 13. Prin definirea tipului, se face doar descrierea unei mulțimi posibile de valori, fără a se cere rezervarea unei zone de memorie corespunzătoare.

Pentru a împiedica tipul respectiv în operații, este necesară declararea uneia sau mai multor variabile de tipul respectiv.

În cadrul programelor, toate definițiile de tip se grupează sub cuvântul cheie **type** și trebuie plasate după definirea constantelor și înaintea declarațiilor de variabile.

9.1. TIPUL SCALAR

În cadrul capitolului 5 au fost prezentate în detaliu patru dintre componentele tipului scalar, grupate sub denumirea de tipuri standard. În continuare, sunt prezentate ultimele două componente.

9.1.1. Tipul enumerare

Mulțimea valorilor unui tip enumerare se definește prin enumerarea identificatorilor ce reprezintă aceste valori. Diagrama de sintaxă pentru tipul enumerare este prezentată în figura 9.5.

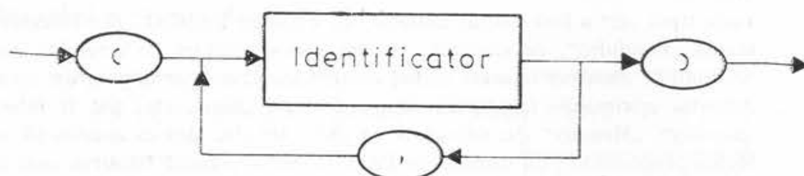


Fig. 9.5

Exemplul 1.

```
type culoare  = (verde, roșu, galben, alb, portocaliu, negru) ;
     zi       = (luni, marți, miercuri, joi, vineri, sâmbătă, duminică) ;
     anotimp  = (primăvară, vară, toamnă, iarnă) ;
```

Exemplu incorect

```
type zi    = (lu, ma, mi, j, vi, s, d) ;
     liber = (s, d)
```

Definirea tipului „liber" este incorectă, deoarece identificatorul „s" este folosit în mod ambiguu.

Prin definirea tipului se precizează atât identificatorul tipului (de exemplu : culoare, zi, anotimp), cât și identificatorii reprezentând valorile posibile (verde, vară, joi etc.). Cei din urmă pot fi folosiți cu rol de constante, mărind astfel claritatea programului (de altfel se mai numesc și „constantele tipului"). Dacă în program există o declarație :

var decor : culoare ;

se pot descrie atribuiri precum :

decor : = verde ;

decor : = roșu ;

care sunt mult mai expresive decât în cazul în care culorile s-ar fi reprezentat numeric și variabila „decor" ar fi fost de tip Integer.

Deoarece mulţimea valorilor unui tip enumerare este ordonată, între elementele sale se pot aplica operatorii relaţionali =, <>, <, <=, >=, >. Numărul de ordine al unui element este determinat de poziţia identificatorului în listă. De asemenea, pot fi folosite funcţiile standard **pred**, **succ** şi **ord**, cu parametru de tip enumerare.

Numărul de ordine corespunzător primului identificator enumerat este 0, în continuare fiind valabilă relaţia :
ord (x) = ord(pred(x))+1

Exemplul 2. Considerând definirea tipurilor din Exemplul 1, atunci :

succ(roşu)	este galben ;
pred(marţi)	este luni ;
ord(verde)	este O ;
vară > primăvară	este True;

În reprezentarea internă, mărimile de tip enumerare apar ca întregul corespunzător numărului de ordine dar, spre deosebire de întregi, între ele nu se pot aplica operatorii aritmetici şi deci, nu trebuie să apară în expresii de calcul.

Observaţie : Tipul boolean este un tip enumerare predefinit, după cum urmează :

```
type boolean=(false, true);
```

Exemplul 3. Să presupunem că am scris următoarea declaraţie de variabilă :
ziua: zi;
unde tipul „zi" a fost definit anterior. În limbajul PASCAL, identificatorul constantă „duminică", de exemplu, nu are nici o legătură cu şirul de caractere 'duminică', deoarece în acest limbaj identificatorii ce apar în program nu au semnificaţie intrinsecă. Identificatorii „duminică", „luni", etc. pot fi înlocuiţi cu „Sunday", „Monday" etc. sau chiar cu „A", „B" etc. fără ca aceasta să afecteze logica programului. Ca urmare, în PASCAL nu e permisă folosirea unor instrucţiuni cum ar fi write(ziua), cu ajutorul cărora ar putea fi tipărit unul dintre cuvintele 'duminică', 'luni' etc., în funcţie de valoarea variabilei ziua. Pentru a obţine o astfel de tipărire ea trebuie programată în mod explicit :

```
case ziua of
      duminică : write('duminică');
      luni     : w'ite('luni');
      marţi    : write('marţi') ;
      miercuri : write('miercuri');
      joi      : write('joi');
      vineri   : write('vineri');
      sâmbătă  : write('sâmbătă')
end.
```

Am menţionat anterior că elementele tipului enumerare pot fi folosite în program cu rol de constante. Iată un exemplu în acest sens :

Exemplul 4. În cadrul următorului fragment de program se citeşte o culoare reprezentată în datele de intrare cu ajutorul caracterelor 'v', 'r', 'g', 'a', 'p', sau 'n', presupunând că dispunem de o variabilă „caracter" de tip char.

```
read(caracter);
case caracter of
      'v' : decor:=verde;
      'r' : decor:=roşu:
      'g' : decor:=galben;
```

```
'a' : decor:=alb;
'p' : decor:=portocaliu;
'n' : decor:=negru
end.
```
De asemenea, se pot scrie astfel de instrucţiuni:
```
for decor:=negru downto roşu do op1;
while (c1 < > c) and b do op1;
if c > alb then c:=pred(c);
```
unde s-a presupus că variabilele c1, c sunt de tip culoare, b este de tip logic iar op1 reprezintă numele generic al unei instrucţiuni.

În concluzie, *utilizarea tipului enumerare conferă programelor scrise în limbajul PASCAL o mai mare claritate şi asigură naturaleţea exprimării* (apropierea de limbajul natural).

9.1.2. Tipul subdomeniu

În anumite cazuri se cunosc limitele între care poate lua valori o variabilă de tip scalar şi, prin urmare, tipul variabilei se poate defini ca un subdomeniu (fig. 9.6) al tipului scalar respectiv. În definiţie se precizează limitele inferioare şi, respectiv superioară ale intervalului. Constantele trebuie să fie de acelaşi tip (tipul scalar pe care se defineşte subdomeniul) şi trebuie să se afle în relaţia :

constanta1 < constanta2

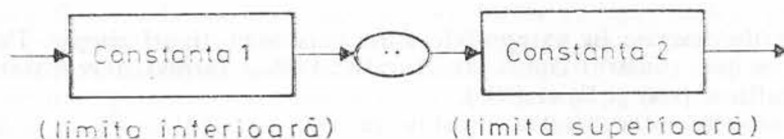


Fig. 9.6.

Nu se pot defini subdomenii ale tipului real. Operatorii (relaţionali, aritmetici) şi funcţiile standard corespunzătoare unui anumit tip scalar sunt valabile pentru orice subdomeniu al său. Utilizarea acestui tip face ca programul să fie mai uşor de citit şi de înţeles. O declaraţie, cum ar fi, de exemplu :

N : 1..10; sau

N : 0..MaxInt ;

stabilind în mod clar posibilele valori ale variabilei N, poate furniza mai multe informaţii decât

N : Integer ;

care ar fi mai indicată în cazul în care nu se cunoaşte mulţimea valorilor posibile ale lui N.

Exemplul 5. Definind iniţial tipurile enumerare:

luna = (ian, feb, mar, apr, mai, iun, iul, aug, sep, oct, nov, dec) ;
zi = (luni, marţi, miercuri, joi, vineri, sâmbătă, duminică);

pot fi declarate ulterior variabile aparţinând unor tipuri subdomeniu :

ZiLucrătoare : luni..vineri ;
LuniDeVară: iun...aug;

Exemplul 6. Programul care urmează totalizează, pe baza înregistrărilor de la o stație meteorologică, numărul de zile ploioase din ultimul sfert de veac.

```
program Ploi;
  const AnInitial=1969;
        AnFinal=1993;
  var An:AnInitial..AnFinal+1;
      ZileCuPloi:0..366;
      Total:0..MaxInt;
begin
      Total:=0;
      An:=AnInitial;
      while An <=AnFinal do
        begin
            write('Nr. de zile ploioase în ',An:4,'?=');
            readln(ZileCuPloi);
            Total:=Total+ZileCuPloi;
            An:=An+1
        end;
      writeln;
      writeln('Total zile ploioase 1969—1993: ', Total:4)
end.
```

9.2. TIPURI STRUCTURATE DE DATE

Tipurile descrise în paragrafele anterioare sunt **tipuri simple**. Pe baza acestora se pot construi tipuri structurate : tablou **(array)**, înregistrare **(record)**, mulțime **(set)** și fișier **(file)**.

Deosebirile între acestea constau în :

— *tipul componentelor ;*

— *metoda de structurare ;*

— *tehnica de selectare* din ansamblu a elementelor componente.

Necesarul de memorie pentru reprezentarea unei structuri rezultă din tipul și numărul componentelor sale și rămâne **constant** pe parcursul execuției programului, motiv pentru care aceste structuri se numesc **structuri statice**.

9.2.1. Tipul tablou (array)

Tabloul este o *structură formată dintr-un număr fix de componente*, toate *de același tip*, numite *elemente*. Selectarea unui element al tabloului se face cu ajutorul numelui variabilei tablou urmat de așa-numitul **indice** cuprins între paranteze pătrate, indice care precizează poziția elementului **în** cadrul structurii.

La definirea unui tablou se precizează :

— *tipul elementelor* (tipul de bază al tabloului) ;

— *tipul permis pentru indici.*

Prin tipul indicelui se fixează implicit și numărul componentelor tabloului, care va fi egal cu cardinalitatea tipului indicelui.

Diagrama de sintaxă simplificată a tipului tablou este prezentată în figura 9.7.

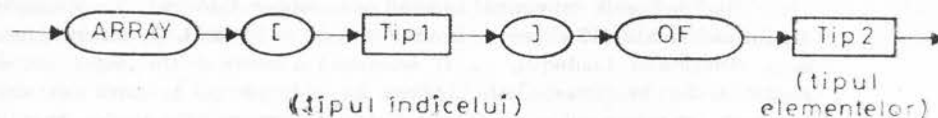


Fig. 9.7.

Tipul indicelui poate fi orice tip scalar cu excepția tipului integer și a tipului real, adică : boolean, char, enumerare, subdomeniu. Tipul elementelor poate fi oricare dintre tipurile permise în limbajul PASCAL.

Elementele unei structuri tablou se mai numesc și **variabile indexate** deoarece pot fi selectate, așa cum am arătat deja, cu ajutorul numelui variabilei tablou și al indicelui care precizează poziția acestora în cadrul structurii.

Diagrama de sintaxă simplificată a unei astfel de variabile este ilustrată în figura 9.8.

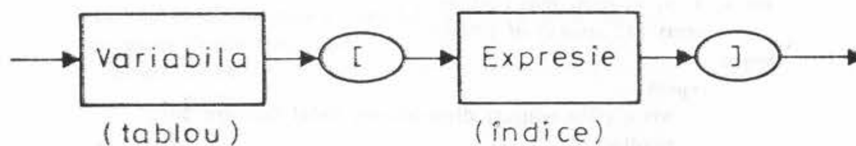


Fig 9.8.

Ca indice se poate folosi orice expresie. Aceasta se evaluează în momentul selectării elementului iar rezultatul trebuie să fie de tipul indicat pentru indice la descrierea tipului tablou, ceea ce presupune încadrarea lui în limitele precizate prin definiție. Îndeplinirea acestor condiții se verifică în mod riguros iar nerespectarea lor se semnalează ca eroare și poate cauza terminarea forțată a programului.

În programare, indicii sunt asemănători în multe privințe cu indicii matematici care în notația convențională se scriu în partea de jos a elementului, fără paranteze : x_i reprezintă, de exemplu, elementul de pe poziția i din șirul (sau vectorul) x, presupus a avea n elemente componente. În PASCAL acest element va fi apelat ca x[i], parantezele fiind necesare datorită limitării posibilităților de reprezentare cu indice inferior a majorității dispozitivelor de intrare.

Exemplul 7. Presupunând că definim tipul enumerare :

Luna=(ian, feb, mar, apr, mai, iun, iul, aug, sep, oct, nov, dec) ;
se pot scrie următoarele declarații de variabile de tip tablou :

```
NrCaracter  : array [char] of 0..MaxInt;
NrLitera    : array ['A'..'Z'] of 0..MaxInt;
Contor      : array [1..4] of integer;
ZileÎnLuna  : array [Luna] of 28..31;
```

Variabila tablou ZileÎnLuna poate fi folosită pentru memorarea zilelor fiecărei luni a anului. După inițializare, ea va avea următoarea formă :

	ian	feb	mar	apr	mai	iun	iul	aug	sept	oct	nov	dec
ZileÎnLuna	31	28	31	30	31	30	31	31	30	31	30	31

De exemplu, valoarea elementului ZileÎnLuna[ian] este 31.

Tipul indicelui determină numărul elementelor tabloului. De exemplu, ZileInLuna va avea 12 elemente. În cazul în care se încearcă, de exemplu, folosirea elementului Contor[0], va fi semnalată o eroare de tip „index out of range", a cărei semnificație este „indicele nu se încadrează în gama de valori permisă", iar programul se va încheia forțat. În scrierea programelor, apariția unor astfel de erori trebuie prevăzută și evitată cu multă grijă.

Cu toate că memoria multor calculatoare este foarte încăpătoare, ea nu este nelimitată ca dimensiune, iar declararea unui tablou cu tipul indicelui 1..100000, de exemplu, este puțin probabil să fie acceptată de către compilator. Majoritatea problemelor de programare pot fi rezolvate fără a face apel la tablouri de dimensiune foarte mare.

Exemplul 8. Program pentru determinarea celui mai mare și a celui mai mic număr dintr-o listă de numere introduse de la tastatură.

```
program MinMax;
const nmax=20;
var n, i, u, v, min, max : integer ;
    A:array [1..nmax] of integer ;
begin
    repeat
        write ('Introduceți dimensiunea listei (maxim 20)');
        readln(n)
    until (n >=2) and (n <=20);
    writeln;
    for i:=1 to n do
      begin
        write('A[', i:2,']=');
        readln(A[i])
      end ;
    min:=A[1];
    max:=min;
    i:=2;
    while i < n do
      begin
        u:=A[i];
        v:=A[i+1];
        if u > v then
            begin if u > max then max:=u;
                  if v < min then min:=v
            end ;
        else
            begin if v > max then max:=v;
                  if u < min then min:=u
            end ;
        i:=i+2
      end
    if i=n then
        if A[n] > max then max: =A[n]
        else if A[n] < min then min :=A[n];
    writeln ('max=', max:6,'min=',min:6)
    end.
```

Deoarece tipul elementelor poate fi orice tip admis de limbajul PASCAL, elementele unui tablou pot fi la rândul lor de tip structură. În particular, dacă ele sunt de tip tablou, se definește un tablou multidimensional, ca în exemplul de mai jos:

var m : **array** [TipIndice1] **of array** [TipIndice2] **of** TipElemente;

Notația m[i][j] indică elementul de pe poziția j (de tip TipElemente) al elementului i al lui m.

Declarația de mai sus se poate prescurta sub forma:

var m : **array** [TipIndice1, TipIndice2] **of** TipElemente;

iar m[i][j] este echivalent cu m[i, j].

Tabloul m poate fi considerat o matrice iar m[i, j] elementul situat pe linia i și coloana j a acesteia. În acest caz este vorba despre un tablou bidimensional (considerând TipElemente orice tip în afară de tipul structură). Dacă TipElemente este un tip structurat se poate ajunge până la un tablou n-dimensional în cadrul căruia un element va fi selectat folosind n expresii indice.

Exemplul 9. Program pentru înmulțirea a două matrici.

```
program ProdusMatriceal;
  const m=4;
        p=3;
        n=2;
  var i:1..m;
      j:1..n;
      k:1..p;
      a:array [1..m, 1..p] of integer;
      b:array [1..p, 1..n] of integer;
      c:array [1..m, 1..n] of integer;
  begin {atribuirea valorilor inițiale pentru a și b}
    for i:=1 to m do
      begin
        for k:=1 to p do
          begin
            write ('a[', i:2,',',k:2,'] =');
            readln (a[i, k])
          end;
        writeln
      end;
    writeln;
    for k:=1 to p do
      begin
        for j:=1 to n do
          begin
            write ('b[', k:2 ,',', j:2,']=');
            readln(b[k, j])
          end;
        writeln
      end;
    writeln;
    {înmulțirea matricilor a și b cu tipărirea rezultatului}
    writeln('Matricea produs:');
    writeln;
```

```
    for i:=1 to m do
      begin
        for j:=1 to n do
          begin
            c[i, j]:=0;
            for k:=1 to p do c[i, j]:=c[i, j]+a[i, k]*b[k, j];
            write (c[i, j]:6)
          end ;
        writeln
      end ;
    writeln
  end.
```

Corespondența dintre noțiunile matematice și tipul tablou se face astfel : vectorii sunt tablouri unidimensionale iar matricile sunt tablouri bidimensionale. Dacă dorim să declarăm vectorul V care conține N elemente și matricile A, B și C de dimensiune M*N (unde M și N sunt constante definite în mod corespunzător) vom scrie :

```
V : array [1..N] of real ;
A, B, C : array [1..M, 1..N] of real ;
```

● **Reguli ce trebuie respectate în cazul tipului tablou :**

Între tablouri de același tip se poate aplica instrucțiunea de atribuire. Se consideră ca două sau mai multe variabile tablou sunt de același tip numai în cazul în care au fost declarate în același loc sau au fost declarate cu ajutorul aceluiași identificator de tip. De exemplu, dacă vom declara :

```
const
  NrElevi=40 ;
  NrMaterii=12
var
  Elev      : 1..NrElevi ;
  Catalog   : array [1..NrElevi] of array [1..NrMaterii] of 0..10 ;
  NoteElev  : array [1..NrMaterii] of 0..10 ;
```

instrucțiunea de atribuire :

```
NoteElev : = Catalog[Elev] ;
```

nu este corectă deoarece cele două variabile tablou au fost declarate separat. Dacă însă se introduce definirea de tip :

```
Note = array [1..NrMaterii] of 0..10 ;
```

și se rescriu declarațiile de variabile astfel :

```
Catalog : array [1..NrElevi] of Note ;
NoteElev : Note ;
```

instrucțiunea de atribuire anterioară va fi considerată ca fiind corectă deoarece ambele tablouri sunt acum de tip Note și va avea ca efect copierea unei linii a tabloului Catalog în tabloul NoteElev.

9.2.2. Tablouri impachetate

Elementele datelor descrise în capitolele anterioare se memorau fiecare în câte o locație de memorie, acesta fiind modul natural sau neîmpachetat (unpacked) de stocare a unor astfel de date.

În momentul definirii unei structuri se poate cere în mod explicit alocarea spațiului de memorie minim necesar reprezentării tuturor valorilor posibile,

caz în care calculul dimensiunii se face fără a ține seama de considerente privind optimizarea accesului la componente. Un astfel de tip se numește **împachetat** și opțiunea se indică prin cuvântul cheie **packed**, care îi precede definiția.

Această metodă este eficientă pentru tipul mulțime și pentru structuri cu componente de tip boolean, char, enumerate și subdomeniu.

Manipularea excesivă a componentelor unei astfel de structuri duce la scăderea vitezei de execuție a programului, motiv pentru care *structurile împachetate se vor folosi mai ales pentru stocarea informației*. Totuși, atunci cînd în program sunt necesare tablouri de dimensiuni mari, este de dorit ca memoria să fie folosită mult mai judicios și drept urmare se preferă comprimarea mai multor elemente într-o aceeași locație. Pentru a ilustra această metodă de utilizare a memoriei vom folosi următoarea **analogie** : să considerăm exemplul unui tablou care conține opt numere întregi formate din câte două cifre zecimale. În mod normal, pentru memorarea sa vor fi necesare opt locații,

20 13 42 25 10 0 3 80

Dacă dispunem de un calculator care poate memora în fiecare locație câte un număr întreg compus din patru cifre zecimale, vom putea comprima câte două elemente ale tabloului în fiecare locație, și întregul tablou va ocupa patru locații :

2013 4225 1000 0380

Dacă dispunem de un calculator care poate memora în fiecare locație câte un număr întreg compus din șase zecimale, pentru memorarea întregului tablou vor fi necesare numai trei locații, o parte a ultimei locații rămânând neocupată.

201342 251000 0380 ??

În fiecare dintre aceste două situații se spune că **tabloul este împachetat (packed)**. Elementele componente ale unui tablou împachetat pot fi accesate, dar această operație este mai dificilă, deoarece necesită „spargerea" informației memorate în elementele ce o compun. Economisirea spațiului de memorie are drept consecință în acest caz încetinirea accesului la componentele individuale.

În limbajul PASCAL, un tablou împachetat se specifică prin folosirea cuvintelor cheie **packed array**. De exemplu :

```
type Admis = boolean ;
var RezultatExamen : packed array [1..NrCandidaţi] of Admis ;
```

Împachetarea acestui tablou poate reduce necesarul de memorie de 4 până la 32 de ori.

În limbajul PASCAL nu există posibilitatea specificării modului în care vor fi împachetate datele, acest lucru fiind realizat în mod automat de către compilator care va alege un aranjament corespunzător, în funcție de dimensiunea elementelor ce trebuie împachetate și de capacitatea locațiilor de memorie ale calculatorului. Folosirea atributului de „împachetat" nu modifică înțelesul programului care va produce aceleași rezultate ca și în cazul în care acest atribut nu este prezent. Principala deosebire constă în eficiența programului: pentru memorarea datelor va fi necesar un spațiu mai redus dar accesarea componentelor individuale va fi mai lentă și va necesita un program în cod mașină de dimensiune mai mare.

• **Şiruri de caractere (varianta standard)**

Un şir de caractere (string) este un *tablou împachetat de tip caracter ce conţine cel puţin un element*. Astfel de şiruri au fost deja folosite ca parametri ai procedurilor **Write**, cum ar fi :

write('suma')

unde şirul de caractere 'suma' este o constantă de tip şir :

packed array [1..4] **of** char

a cărei lungime este egală cu 4 (numărul de caractere din şir).

Aşa cum rezultă din exemplul următor, **limbajul PASCAL permite :**

— *definirea de constante şir* ;

— *definirea de tipuri şir* ;

— *declararea de variabile şir*.

Exemplul 10.

```
const
   copil    ='Eugen     ';
   DimMax  = 10;
type
   Nume=packed array [1..12] of char ;
var
   NumeCopil:packed array [1..DimMax] of char;
   Mama, Tata:Nume;
În continuare se pot scrie astfel de instrucţiuni :
NumeCopil:=copil;
Mama      :='Maria       ';
Tata      :='George      ';
```

Observaţie : Şirul trebuie să aibă exact aceeaşi lungime ca şi variabila şir căreia îi este atribuit. De exemplu, variabilei NumeCopil îi putem atribui numai un şir de 10 caractere.

Variabilele de tip şir sunt deseori folosite pentru memorarea şi manevrarea cuvintelor şi a textelor. În exemplul prezentat, în variabilele Mama şi Tata se pot memora nume formate din cel mult 12 caractere.

Mama	'M'	'A'	'R'	'I'	'A'	" "	" "	" "	" "	" "	" "	" "
Tata	'G'	'E'	'O'	'R'	'G'	'E'	" "	" "	" "	" "	" "	" "

Această reprezentare presupune că variabilele menţionate sunt tablouri de caractere. Dacă dorim să le reprezentăm ca variabile şir, vom scrie :

Mama : 'MARIA   '

Tata: 'GEORGE   '

În exemplul prezentat, cele două nume au mai puţin de 12 caractere semnificative şi sunt completate cu spaţii. Aceasta este o metodă uzuală numită *„umplere cu spaţii"*. Numele propriu-zis este plasat în partea stângă iar spaţiile suplimentare sunt adăugate în partea dreaptă. Se spune că numele a fost memorat aliniat la stânga, aceasta fiind convenţia uzuală pentru cuvinte şi texte. În mod analog, numele poate fi memorat şi aliniat la dreapta în cadrul şirului, dar această convenţie este specifică pentru scrierea numerelor într-un fişier text, după cum se va vedea în capitolul 12.

Exemplul 11. Să presupunem că în variabila şir NumeCopil a fost memorat aliliniat la stânga numele :
'DAN ION '

Dorim să elaborăm un fragment de program care să scrie acest nume centrat pe o linie a cărei dimensiune este DimLinie=60 coloane.

O schiţă a acestui program este următoarea :

— se determină lungimea reală a numelui (Lung);

— se scrie numele, precedat de ((DimLinie — Lung) **div** 2) spaţii.

Lungimea reală a numelui se poate determina prin căutarea de la dreapta la stânga a primului caracter diferit de spaţiu (o căutare liniară).

```
program CentrareText;
const Spaţiu=' ';
      DimMax=10;
      DimLinie=60;
var   Lung    :0..DimMax;
      Umplere:boolean;
      NumeCopil :packed array [1..DimMax] of char;
begin
      NumeCopil:='DAN ION    ';
      {se determină lungimea numelui}
      Lung:=DimMax;
      Umplere:=true;
      while Umplere and (Lung > 0) do
            if Nume Copil[Lung] < > Spaţiu then
                  Umplere:=False
            else
              Lung:=Lung—1;
      {se scrie numele, precedat de numărul necesar de spaţii}
      writeln(spaţiu:(DimLinie—Lung) div 2, NumeCopil)
end.
```

În cazul nostru, numele va apare în forma

DAN ION

în care numele este precedat de 26 de spaţii.

Observaţie : Limbajul PASCAL standard permite scrierea unui şir complet cu ajutorul procedurii Write (vezi exemplul anterior), dar nu permite citirea unui şir complet cu instrucţiunea Read, deoarece nu s-ar putea preciza numărul de caractere ce trebuie citite. De aceea, programatorul trebuie să gândească şi să scrie în cadrul codului această operaţie într-un mod asemănător exemplului următor.

Exemplul 12. Fragment de program care realizează citirea caracterelor în variabila de tip şir:

Şir : **packed array** [1..L] **of** char ;

Se vor citi exact L caractere sau, în cazul în care este întâlnit sfârşitul liniei curente a intrării, variabila Şir va fi completată la dreapta cu spaţii. Folosind variabila :

Poziţie:1..L;

se poate scrie:

for Poziţie:=1 **to** L **do**

```
if EOLn then
      şir[Poziţie]:=Spaţiu
   else
      Read(Şir [Poziţie]);
```

În cazul în care L = 9 şi linia de intrare are forma :

I O N E S C U   T U D O R

rezultatul va fi 'IONESCU T'

Dacă citirea se face începând din poziţia indicată prin săgeată :

I O N E S C U   T U D O R

rezultatul va fi 'TUDOR   '

Ordonarea alfabetică a cuvintelor este o operaţie des întâlnită în practică. De exemplu, în cazul cărţilor de telefon, numele abonaţilor pot să apară în forma :

DINU
DUMITRU
ENESCU
FLOREA
LUPAŞ
LUPU

fiind ordonate în raport cu prima literă ; în cazul în care prima literă este aceeaşi, numele se ordonează în raport de a doua literă şi aşa mai departe. În cadrul calculatoarelor ordonarea este definită pentru toate caracterele (nu numai litere) astfel încât această ordonare lexicografică poate fi generalizată şi în cazul şirurilor.

În capitolul 5 s-a arătat că se pot compara caractere individuale folosind operatorii relaţionali =, <>, <, >, <=, >=, >. Şi în cazul şirurilor complete care au aceeaşi lungime se poate proceda în mod asemănător. De exemplu, dacă valoarea variabilei şir NumeFamilie este 'Lupu', atunci toate expresiile care urmează au valoarea logică true (adevărat).

```
NumeFamilie = 'Lupu      '
NumeFamilie < 'Lupaş     '
NumeFamilie < 'Rotaru    '
NumeFamilie <>'Dinu      '
NumeFamilie>='Dumitru    '
NumeFamilie<='Lupu       '
```

● **Reguli pentru şiruri**

Compatibilitatea atribuirii : şirul trebuie să aibă exact aceeaşi lungime ca a variabilei şir căreia îi este atribuit. (Regula este ceva mai puţin restrictivă decît regula analogă pentru alte tablouri).

Comparaţii : se pot compara numai şiruri care au exact aceeaşi lungime.

Observaţie : Pentru a beneficia complet de posibilităţile limbajului PASCAL de a folosi constantele de tip şir şi de a compara în mod direct şiruri, se recomandă folosirea tablourilor caracter împachetate. Împachetarea comprimă aceste tablouri de aproximativ patru ori şi măreşte viteza de realizare a atribuirilor şi a comparaţiilor şirurilor tot de atâtea ori.

Celelalte tipuri structurale de date vor fi tratate în cadrul capitolului 12, după parcurgerea secţiunilor referitoare la subprograme, întrucât exemplele asociate folosesc noţiuni legate de funcţii şi proceduri şi prezintă un grad mai mare de complexitate.

9.2.3. Tipul string (şir de caractere)

Pentru manipularea şirurilor de caractere, în varianta standard a limbajului se pot folosi numai tablourile împachetate cu elemente de tip char. Limitările şi deficienţele de manipulare ale acestora sunt eliminate în versiunile Turbo-PASCAL prin includerea ca tip predefinit a tipului **string**. Cu toate că lucrarea de faţă este axată pe descrierea elementelor limbajului standard, în acest paragraf (ca o excepţie) sunt descrise principalele caracteristici ale tipului string, tip larg folosit de către utilizatorii sistemelor de calcul compatibile IBM-PC. De asemenea, sunt descrise pe scurt şi principalele funcţii asigurate de Turbo-PASCAL pentru prelucrarea şirurilor.

Un **tip string** este un *tip special de vector, cu elemente de tip char, care memorează, în afara caracterelor din şir, şi lungimea respectivului şir de caractere, adică numărul de caractere existente efectiv în şir.*

La definirea tipului string sau la declararea explicită a unei variabile de acest tip se precizează de cele mai multe ori şi valoarea maximă pe care o poate lua această lungime. O valoare de tip string poate fi orice şir de caractere cu lungime cuprinsă între zero şi lungimea specifică în definiţia tipului. Şirul de lungime zero, denumit **şir vid** se reprezintă prin constanta ''.

Diagrama de sintaxă a tipului **string** este prezentată în figura 9.9.

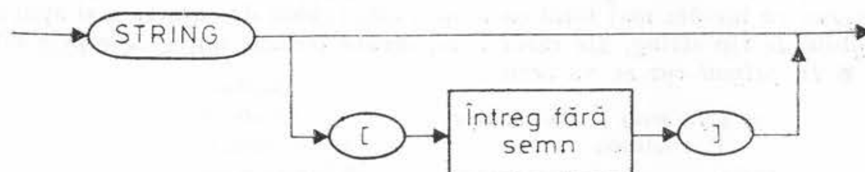


Fig. 9.9.

Dacă în definirea tipului apare numai cuvîntul-cheie **string**, se presupune că dimensiunea implicită este 255. Dacă apare în cadrul definiţiei, dimensiunea este specifică cu ajutorul unui întreg fără semn ce poate lua valori între 1 şi 255. Limitarea la 255 de caractere se datorează modului de reprezentare efectivă a valorilor de tip string. Din punctul de vedere al memorării, definiţia :

şir1 = **string** [10] ;

este echivalentă cu :

şir2 = **array** [0..10] **of** char ;

în care elementul cu indice zero este folosit pentru memorarea unui caracter cu cod egal cu lungimea efectivă a şirului de caractere. Deoarece codul unui caracter este un întreg aparţinând intervalului 0..255, apare ca evidentă necesitatea limitării dimensiunii şirului la valoarea 255.

Elementele componente ale unei variabile de tip string pot fi accesate la fel ca și elementele unui tablou de caractere, conform diagramei de sintaxă prezentate în figura 9.10.

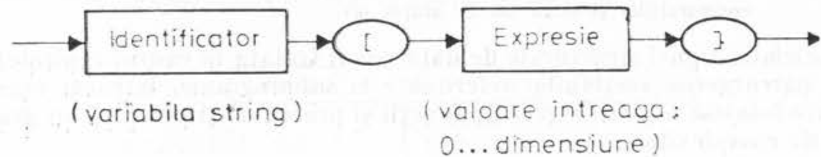


Fig. 9.10.

Exemple :

```
var Mesaj: string [12];
begin
    ...
    Mesaj [1]:='S';
    Mesaj [2]:='t';
    Mesaj [3]:='0';
    Mesaj [4]:='p';
    ...
```

```
var i:0..12;
    Mesaj: string [12];
    car: char;
begin
    ...
    Mesaj [i]:=car;
    ...
    write ('Lungime:', Mesaj [0]);
    ...
```

- **Operații cu șiruri de caractere**

Tipurile string sunt compatibile între ele, atât ca operanzi în expresii, cât și la atribuire. Tipul char este compatibil cu tipurile string (în orice context în care poate să apară o valoare de tip string se poate utiliza și o valoare de tip char, reciproca nefiind însă valabilă).

Un **avantaj important** al tipului **string** este acela că *procedurile standard de scriere și citire admit argumente de tip string completate, de orice dimensiune.* În plus, programatorul nu trebuie să se îngrijească de completarea la dreapta cu blancuri a șirului ca în cazul tablourilor împachetate de caractere, din varianta standard a limbajului. Pentru a exemplifica această afirmație, să presupunem că lucrăm mai întâi cu o variabilă tablou de caractere și apoi cu o variabilă de tip string, ale căror componente trebuie inițializate prin citire.

- *În primul caz* se va scrie :

```
var șirA: array [1..10] of char;
    i   : integer;
begin
    writeln ('Introduceți șirul de caractere:');
    for i =:1 to 10 do
        read (șirA[i]);
    readln;
    write('Șirul de caracter citit este:');
    writeln(șirA)
end.
```

- *În cel de-al doilea caz*, programul se simplifică :

```
var șirB:string [20];
begin
    writeln('Introduceți șirul de caractere:');
    readln(șirB);
    write('Șirul de caractere citit este:');
    writeln(șirB)
end.
```

Valorile de tip string complete pot fi comparate direct cu ajutorul oricărui operator relaţional, ordonarea acestor valori fiind de tip lexicografic şi extinsă la întreg setul de caractere utilizat, pe baza codurilor numerice asociate caracterelor respective, de exemplu :

'x' < 'xyz' 'x2' < 'xa' 'xy' > 'xxy'

Observaţie : în cadrul operaţiilor de citire şi de atribuire a valorilor de tip string complete trebuie avut grijă ca valorile citite sau atribuite să nu conţină mai multe caractere decât dimensiunea variabilei ce le va îngloba deoarece, în caz contrar, caracterele în exces sunt ignorate.

Exemplul 13.

```
var x:string [8];
begin
     ...
     x:='1234567890';
     writeln(x)
end
```

În finalul execuţiei fragmentului de program anterior se va afişa pe ecran valoarea
12345678
caracterele '9' şi '0' neputând fi memorate în variabila x.

Un alt tip de prelucrare asupra şirurilor de caracter este **concatenarea** care realizează construirea unui nou şir din două sau mai multe şiruri. În acest scop se poate folosi operatorul binar + sau funcţia predefinită **concat**. Operanzii pot fi variabile sau constante de tip string sau char iar rezultatul este de tip string. Dacă dimensiunea rezultatului depăşeşte valoarea 255, el este trunchiat după cel de-al 255-lea caracter.

Exemplul 14.

```
var   rezultat : string;
          a, b  : string [4];
begin
     ...
     readln(a);
     readln(b);
     rezultat:=a+b;
     writeln(rezultat);
     ...
end.
```

Datele şi rezultatele afişate pe ecran vor fi, de exemplu :
Test
are
Testare

• Funcţii şi proceduri predefinite pentru prelucrarea datelor de tip string.

Funcţia concat : admite *doi sau mai mulţi parametri de tip string* sau char şi furnizează ca rezultat şirul obţinut prin concatenarea valorilor acestora (are acelaşi efect ca şi operatorul binar +).

Exemplul 15.

```
rezː=a+b;               este echivalent cu rez:=concat (a,b);
```

Funcţia length : *admite ca parametru un string* şi furnizează ca rezultat lungimea acestuia.

Exemplul 16.

writeln(length('exemplu'));

va avea ca rezultat tipărirea valorii 7

Observaţie : deoarece lungimea şirului de caractere memorat într-o variabilă x de tip string poate fi obţinută şi prin evaluarea expresiei ord(x[0]), expresia următoare are valoarea logică True :

length(x)=ord(x[0])

Funcţia pos : este folosită pentru a preciza dacă un şir apare sau nu ca subşir al altui şir. Admite doi parametri : *şirul căutat* şi *şirul în care se efectuează căutarea*. Funcţia furnizează ca rezultat o valoare de tip întreg, cuprinsă între zero şi lungimea şirului în care se caută. Valoarea zero apare în cazul în care şirul căutat nu a fost găsit. Dacă şirul căutat există, valoarea rezultatului precizează poziţia în care a fost depistată prima apariţie a şirului căutat.

Exemplul 17.

pos('ce','cercetare') are valoarea 1

pos('ea','cercetare') are valoarea 0

Observaţie : funcţia pos detectează întotdeauna *prima apariţie a şirului căutat*. Următoarele apariţii trebuie căutate într-o copie a şirului de căutare din care se exclude partea ce se încheie după prima apariţie.

Funcţia copy : realizează copierea unui subşir dintr-un şir, având ca parametri ; *şirul din care se copiază*, *poziţia* (indicele) caracterului din şir cu care începe copierea şi *numărul de caractere copiate*. Rezultatul funcţiei copy este de tip string.

Exemplul 18.

copy('cercetare',4,3) are valoarea 'cet'

copy('cercetare',9,2) are valoarea 'e'

copy('cercetare',11,3) are valoarea ' '

Observaţie : se copiază numărul maxim posibil de caractere. Lungimea rezultatelor apelului copy(s, i, n) este dată de formula :

minim(n, max(0, length (s)— i+1)).

Procedura delete : realizează ştergerea unui subşir din cadrul unui şir. Admite trei parametri : *şirul modificat* (o variabilă de tip string), *poziţia* în cadrul şirului a primului caracter din subşirul ce trebuie şters şi *numărul de caractere ce trebuie şterse* (adică lungimea subşirului ce trebuie şters).

Observaţii :

1. Prin ştergere lungimea şirului prelucrat se reduce ;
2. Nu se pot şterge mai multe caractere decât există efectiv între începutul subşirului şters şi sfârşitul şirului din care se şterge. Un apel de forma : delete (s, i, n) nu are efect dacă i > length (s).

Procedura insert : asigură inserarea unui subşir în cadrul unui şir. Admite trei parametri : *subşirul ce trebuie inserat, şirul în care se face inserarea şi poziţia în şir de la care începe inserarea.* Primii doi parametri sunt de tip string, ultimul de tip integer. Primul şi ultimul parametru sunt transmişi prin valoare (pot fi expresii, constante, variabile) iar cel de-al doilea parametru este transmis prin referinţă (trebuie să fie o variabilă de tip string).

Observații

1. Prin inserare, lungimea șirului prelucrat crește dar nu poate depăși dimensiunea maximă declarată pentru respectivul șir. Altfel, ultimele caractere ale rezultatului se pierd ;
2. Dacă se încearcă inserarea în afara șirului, operația se transformă în concatenare.

Exemplul 19. Procedura pentru înlocuirea aparițiilor unui subșir printr-un alt subșir.

```
procedure Inlocuire (var s:string; vechi, nou:string);
   var k, lungv:integer;
   begin
      lungv:=length(vechi) ;
      k:=pos(vechi, s) ;
      while k > 0 do
         begin
            delete (s, k, lungv);
            insert (nou, s, k) ;
            k:=pos(vechi, s)
         end ;
   end.
```

Dacă vechiul șir este un subșir al celui nou, procedura nu va funcționa corect deoarece, în acest caz, va apare o buclă infinită.

Procedura val : efectuează conversia unui șir de caractere într-o valoare numerică (întreagă sau reală). Admite trei parametri : *șirul de caractere ce trebuie convertit, variabila de tip numeric în care se memorează rezultatul conversiei și o variabilă de tip întreg* căreia i se atribuie o valoare indicatoare a modului în care s-a terminat conversia (valoarea zero arată că nu s-au detectat erori, în timp ce o valoare mai mare ca zero precizează poziția din șirul sursă în care a fost detectată o eroare). Forma generală de apel : val (șir, VariabilăNumerică, VariabilăIntreagă).

Procedura str : realizează conversia unei valori numerice întregi sau reale într-un șir de caractere. Admite doi parametri : *valoarea numerică* (constantă, variabilă sau expresie de tip întreg sau real, urmată eventual de o specificare a formatului de tipărire, la fel ca în cazul parametrilor numerici ai procedurilor write și writeln) și *variabila de tip string*, în care se memorează rezultatul conversiei.

Exemple

str(18.7:5:2,șir)	memorează în șir valoarea '18.70'
str(6+5,șir)	memorează în șir valoarea '11'

Observație : procedura str se folosește mai ales atunci când niciuna dintre formele de tipărire accesibile prin intermediul procedurilor write sau writeln nu este convenabilă pentru tipărirea anumitor valori numerice (situația apare cu precădere la alinierea rezultatelor numerice în tabele). În astfel de cazuri se folosește următoarea secvență de prelucrări :

1. Se convertește valoarea numerică în șir de caractere ;
2. Se prelucrează șirul de caractere, aducându-l la forma dorită ;
3. Se tipărește șirul rezultat.

EXERCIȚII

1. Considerând tipul definit prin enumerarea (roșu, alb, maro, gri) să se stabilească ce valori reprezintă:

a. pred(ord(alb));
b. ord(pred(alb));
c. chr(ord('A')+ord(succ(maro)));
d. ord('Z')—ord('A')—ord(maro);
e. succ(chr(ord('A')+3));

2. Presupunând că sunt declara e următoarele variabile:

```
var
  i:integer;
  j:1..10;
  c:char;
  s:(masculin, feminin);
```

Să se precizeze care dintre următoarele instrucțiuni **for** sunt corecte:

a. **for** j:=1 **to** i **do**...
b. for s:=masculin **to** feminin **do**...
c. **for** s:=0 **to** 1 **do**...
d. **for** c:='A' **to** 'D' **do**...

3. Sunt corecte următoarele declarații de tablou?

a. **var** t : **array** [integer] **of** char ;
b. **var** sir : **array** [1.5...1.8] **of** integer ;

4. Scrieți un program care așază în ordine inversă componentele unui tablou A de tip **array** [1..10] **of** integer.

5. Găsiți erorile din următoarea secvență de program și rescrieți-o corect. Precizare: secvența de program ar trebui să calculeze elementul maxim al tabloului citit.

```
type vector : array [1..n] of real ;
var max: real;
    i : integer ;
begin
  for i = 1 to n do
    write('a[',i,']);
    read(a(i));
  max:=a(1);
  for i = 1 to n do
    if a(1) > max then max=a(i)
end
```

6. Scrieți un program care numără cuvintele și propozițiile unui text introdus de la tastatură. Textul se termină prin introducerea caracterului „sfârșit de fișier". Cuvintele sunt separate prin virgulă și spațiu iar propozițiile se termină cu unul dintre caracterele '.', '!', '?.' Un cuvânt poate începe printr-o literă sau printr-o cifră iar în interiorul cuvintelor se acceptă doar: literele, cifrele și '—'.

7. Considerând că de la mediul de intrare se introduc valorile temperaturilor măsurate, din oră în oră, în luna august precum și cantitățile zilnice de precipitații din această lună, să se scrie programul care afișează:

— temperatura maximă (cu ziua și ora asociată);
— temperatura minimă (cu ziua și ora asociată);
— lista zilelor, ordonată descrescător în funcție de cantitatea de precipitații;
— media precipitațiilor zilnice în luna august.

8. Pentru o matrice cu N linii și M coloane ($1 < N, M < 10$) să se scrie un program care afișează liniile conținând K elemente nule ($0 < K < = M$). Se va afișa un mesaj în situația în care nici o linie nu conține exact K elemente nule.

9. Să se tipărească echivalentul în forma literală al unei valori reale cu maximum 6 cifre pentru partea întreagă și maximum 2 cifre pentru partea fracționară. De exemplu, 24.3 va fi tipărit ca „douăzeci și patru și trei zecimi".

10. Scrieți un program care, operând asupra unei matrici pătrate, întoarce ca rezultat una dintre valorile:

0 — dacă marticea este simetrică față de diagonala principală;
1 — dacă matricea este superior triunghiulară;
2 — dacă matricea este inferior triunghiulară;
3 — dacă matricea are toate elementele nule.

11. Să se scrie un program care, primind un șir X de numere întregi cu N elemente, neordonate, și o valoare întreagă V decide dacă V se află sau nu în șir. În caz afirmativ, tipărește toate pozițiile în care se află valoarea V. În caz negativ, tipărește un mesaj corespunzător.

12. Se dau două șiruri de numere întregi: X cu NX elemente și Y cu NY elementei $NX > NY$. Să se decidă dacă Y este un subșir al lui X, adică dacă există un număr K astfel, încât:

$X_K = Y_1$
$X_{K+1} = Y_2$
...
$X_{K+NY-1} = Y_{NY}$

În caz afirmativ se va tipări valoarea lui K.

13. Se dau două matrice de numere întregi: A cu MA linii și NA coloane și B cu MB linii și NB coloane, astfel încât $MA > = MB$ și $NA > = NB$. Să se decidă dacă B este o submatrice a lui A, adică dacă există K, L astfel încât: $A_{K+I-1,\ L+J-1} = B_{I,J}$ cu $I = 1, MB$ și $J = 1, NB$. În caz afirmativ se vor tipări K și L.

14. Scrieți un program care normalizează un vector dat, V, de dimensiune dată, N, adică împarte fiecare componentă a vectorului prin valoarea absolută maximă depistată prin explorarea valorilor absolute ale tuturor componentelor.

15. Dintr-o matrice dată A, să se listeze valorile tuturor punctelor „șa", împreună cu poziția lor. Un element $A_{i,j}$ este punct „șa" dacă el este elementul minim din linia i și în același timp elementul maxim din coloana j.

CAPITOLUL 10

SUBPROGRAME

10.1. FUNCŢII

Cu toate că limbajul de programare PASCAL pune la dispoziţia utilizatorului un număr de funcţii standard, cum sunt funcţiile sin, exp, ord, abs etc., există foarte multe situaţii în care sunt necesare alte funcţii ce nu fac parte din acest set. De exemplu, limbajul PASCAL nu furnizează funcţii standard pentru calcularea tangentei unui unghi sau pentru a stabili dacă un număr este sau nu prim etc.

Din moment ce nu putem pretinde unui limbaj de programare să asigure existenţa unei funcţii standard pentru orice cerinţă posibilă, este necesar să dispunem de o modalitate de a crea funcţii după dorinţa programatorului. În limbajul PASCAL, o astfel de facilitate este declaraţia de funcţie.

Pornim din nou de la un exemplu :

Exemplul 1. Program pentru calculul combinărilor

$$C_n^r = \frac{n!}{r!(n-r)!},$$

(presupunând că n, r şi (n — r) au valori pozitive, n >= r).

• *Varianta 1.*

— citeşte n,r

— calculează f1=n!

— calculează f2=r!

— calculează f3=(n—r)!

— $C \leftarrow \frac{f1}{f2*f3}$

— tipăreşte C

```
program Combinari1;
var
      n,r,i:integer;
```

```
    c, f1, f2, f3: real ;
begin
    readln(n,r);
    f1:=1;        for i:=1 to n do f1:=f1*i;
    f2:=1;        for i:=1 to r do f2:=f2*i ;
    f3:=1;        for i:=1 to n—r do f3:=f3*i;
    c:=f1/(f2*f3);
    writeln('Rezultat=', c:8:0);
end.
```

Acelaşi lucru se poate scrie, mai compact, folosindu-se funcţia Factorial, definită în cele ce urmează.

● *Varianta 2.*

```
Program Combinari2;
var
    n, r:integer;
    c:real;
function Factorial (x:integer):real;
    var
        i:integer;
        f:real ;
    begin
        f:=1;
        for i:=1 to x do f:=f*i;
        Factorial:=f;
        end ;
begin
        readln(n,r);
        c:=Factorial(n)/(Factorial(r)*Factorial(n—r));
        writeln('Rezultat=',c:8:0)
end.
```

În exemplul precedent se întâlnesc elementele definitorii pentru lucrul cu funcţii :

— declararea funcţiei în secţiunea declarativă a programului ;

— apelul de funcţii (referinţe la funcţii) în interiorul programului ;

— parametri formali, utilizaţi la declararea funcţiei, parametri ce vor fi înlocuiţi la apel cu parametrii actuali.

Observaţii

1. Pentru a scrie o funcţie trebuie precizate :

— numele funcţiei precedat de cuvântul rezervat **function** ;

— parametrul (sau parametrii) săi formali, împreună cu tipul acestora ;

— tipul rezultatului obţinut prin execuţia funcţiei („întors", „returnat" de funcţie) ;

— instrucţiunile ce specifică modul în care se calculează rezultatul funcţiei pe baza valorilor parametrilor actuali.

Sintetizând, se pot trasa următoarele **diagrame de sintaxă** :

— *declararea unei funcţii* (fig. 10.1) ;



Fig. 10.1.

— *antet funcție* (fig. 10.2):

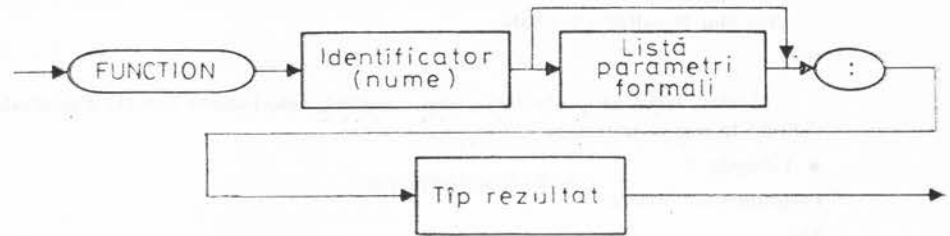


Fig. 10.2.

— *lista parametrilor formali* (fig. 10.3);

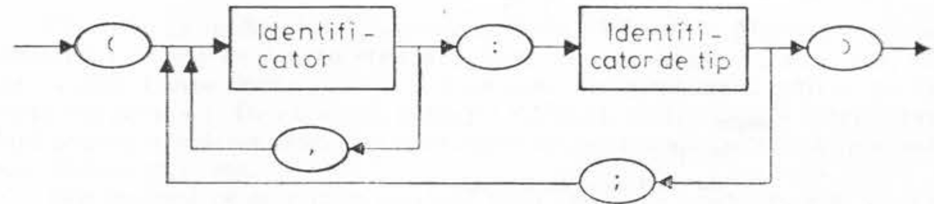


Fig. 10.3.

— *apelul unei funcții* (fig. 10.4):

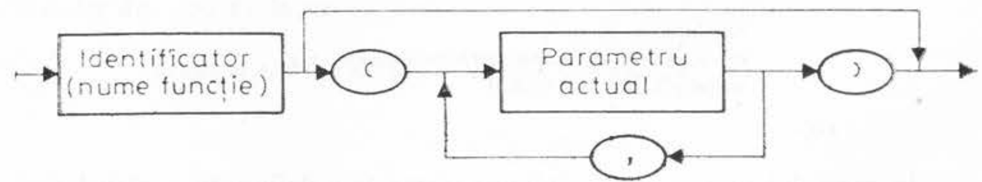


Fig. 10.4.

— *parametru actual* (fig. 10.5):

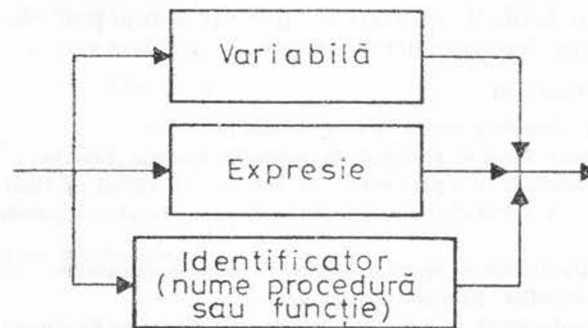


Fig. 10.5.

2. Apelarea unei funcţii nu este o instrucţiune de sine stătătoare ; ea trebuie inclusă ca operand în cadrul expresiilor.

3. **Parametrii formali** sunt disponibili numai în interiorul funcţiei. La apel, numărul **parametrilor actuali** trebuie să fie identic cu cel al parametrilor formali. Valorile parametrilor actuali se atribuie parametrilor formali exact în ordinea precizată la definirea funcţiei. Teoretic, există posibilitatea ca o funcţie să nu aibă parametri formali (ea va întoarce mereu acelaşi rezul'at) !

4. Atât tipul parametrilor cât şi tipul rezultatului pot fi standard sau pot fi definite anterior. Rezultatul funcţiei este reprezentat printr-o **unică valoare** (nu poate fi de tip structurat, de exemplu tabloul).

5. În corpul funcţiei trebuie să existe cel puţin o instrucţiune de atribuire prin care să se transmită numelui funcţiei valoarea rezultatului.

6. O funcţie poate avea ea însăşi propria-i parte declarativă, în care sunt definite constante, tipuri, **variabile locale** şi chiar şi alte funcţii, toate acestea fiind disponibile numai în interiorul funcţiei (nu sunt „văzute" (utilizabile) în programul principal, sau în funcţii exterioare). Prin urmare, o funcţie este ea însăşi un program (care „întoarce" o unică valoare), aşadar se poate numi **subprogram**.

7. **Variabilele globale**, declarate în programul principal pot fi utilizate în interiorul oricărei funcţii asigurând în anumite situaţii un alt mijloc pentru transmiterea rezultatelor.

Exemplul 2.

```
function Putere (x:real, n:integer):real;
    var
      Temp:real;
      P:1..MaxInt;
begin
      Temp:=1.0;
      for P:=1 to abs(n) do
          Temp:=Temp*x;
      if n >= 0 then
          Putere:=Temp
      else
          Putere:=1.0/Temp
end.
```

În exemplul anterior a fost necesară folosirea a două variabile locale, Temp şi P. Variabila *P* este folosită drept contor al unui ciclu **for** iar limbajul PASCAL nu permite utilizarea în acest scop a unei variabile globale. Variabila Temp ar fi putut fi declarată ca variabilă globală dar în acest caz ar fi crescut posibilitatea apariţiei unor efecte imprevizibile în programul apelant.

Observaţie : Încercarea de a elimina variabila de lucru Temp din cadrul funcţiei şi de a scrie :

```
Putere:=1 .0;
for P:=1 to abs(n) do
    Putere:=Putere*n;
```

nu este corectă deoarece Putere nu este o variabilă şi nu poate fi folosită în membrul drept al unei instrucţiuni de atribuire ce face parte din funcţia cu acelaşi nume ; numele funcţiei serveşte doar la returnarea valorii funcţiei şi poate fi folosit în mod corect printr-o instrucţiune, cum ar fi :

```
Putere:=1 .0/Temp;
```

în care numele funcţiei se află în membrul stâng al instrucţiunii de atribuire. Pentru detalii suplimentare, se recomandă parcurgerea capitolului următor (subprograme recursive).

Exemplul 3. Utilizarea funcţiei Putere în cadrul unui program.

```
program PuteriReale;
var v:real ;
    m, n, P : integer;
function Putere (x:real; n:integer):real;
  var Temp:real ;
      P:1..MaxInt ;
begin
      Temp :=1.0;
      for P:=1 to abs(n) do
        Temp : = Temp*x;
      if n >= 0 then
        Putere:=Temp
      else
        Putere:=1.0/Temp
  end; {Putere}
begin
      read(v, m, n);
      for P:=m to n do
        writeln(v,' la puterea ', P:1,'=', Putere(v, p))
end.
```

Variabilele globale şi variabilele locale pot avea acelaşi nume, fără ca acest lucru să aibă influenţă asupra valorilor lor (de exemplu, variabilele notate cu P în programul PuteriReale). În interiorul funcţiilor se recomandă cu predilecţie folosirea variabilelor locale şi nu a celor globale, deoarece în acest mod scade gradul de interacţiune dintre programul principal şi funcţie şi se minimizează riscul apariţiei erorilor.

În concluzie, sunt prezentate **etapele de execuţie a unei funcţii** (asigurate de către sistemul de calcul în momentul rulării) :

1. Se creează o locaţie de memorie pentru rezultatul funcţiei. Valoarea iniţială a acesteia este nedefinită ;

2. Se pun în corespondenţă, de la stânga la dreapta, parametrii actuali ai funcţiei cu parametrii formali ; deci, fiecare parametru formal se înlocuieşte cu exact un parametru actual ;

3. Se creează câte o locaţie de memorie pentru fiecare parametru formal şi i se atribuie acesteia valoarea parametrului actual corespunzător. (Parametrii actuali trebuie să corespundă din punctul de vedere al compatibilităţii atribuirii cu tipul parametrilor formali corespunzători) ;

4. Se creează câte o locaţie de memorie pentru fiecare variabilă locală a funcţiei. Valoarea iniţială a acesteia e nedefinită ;

5. Se execută instrucţiunile cuprinse în corpul funcţiei ;

6. În final se eliberează toate locaţiile de memorie ce fuseseră rezervate pentru rezultatul funcţiei, parametrii formali şi variabilele locale. Valoarea rezultatului funcţiei este utilizată în cadrul expresiei ce include apelul acesteia.

10.2. PROCEDURI

În cadrul paragrafului precedent s-a arătat că o funcție este o parte de program „de sine stătătoare" (conține detalii ce nu interesează utilizatorul : definiri de constante, declarații de variabile, definiri de „subfuncții" etc., importante pentru acesta fiind numai lista de parametri și tipul rezultatului). De exemplu, pentru a putea fi folosite, nu interesează detaliile interne ale funcțiilor sin, cos etc.

Referirea la o funcție poate să apară în cadrul programului principal în membrul drept al unei instrucțiuni de atribuire, ca operand într-o expresie sau ca parametru actual într-o listă de parametri pentru un alt apel de funcție sau procedură.

În limbajul PASCAL, funcțiile sînt asemănătoare funcțiilor din matematică. Totuși, funcțiile prezintă o anumită **limitare** : *au ca rezultat o singură valoare*. Deoarece de multe ori este necesară obținerea ca rezultat a mai multor valori sau ca acesta să fie de tip structurat, limbajul PASCAL permite utilizarea unui al doilea tip de subprogram, **procedura.**

În urma unei apelări (invocări) cu ajutorul **instrucțiunii procedură**, procedurile produc zero, unul sau mai multe rezultate.

Spre deosebire de funcții, procedurile furnizează rezultate prin intermediul parametrilor și nu prin numele asociat. Până acum s-au folosit deja anumite proceduri speciale, cu număr variabil de parametri, cum ar fi Read și Write. Procedurile uzuale, însă, au un număr fix de parametri, de tip bine precizat.

În figura 10.6 este prezentată diagrama de sintaxă a declarației de procedură.

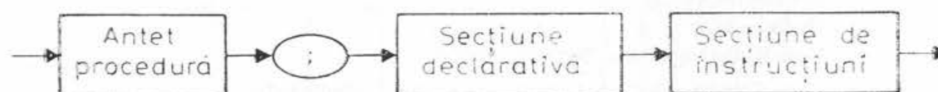


Fig. 10.6.

Figura 10.7 prezintă diagrama de sintaxă pentru Antet procedură.

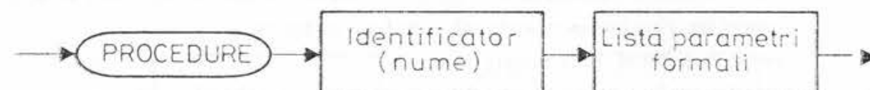


Fig. 10.7.

Instrucțiunea procedură are diagrama de sintaxă din figura 10.8.

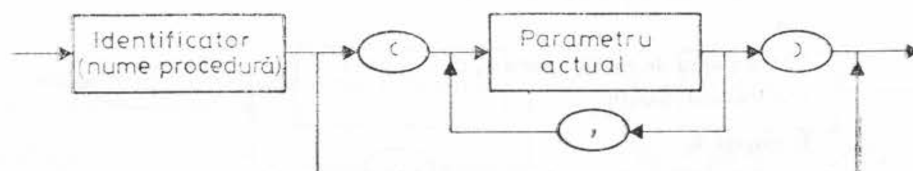


Fig. 10.8.

La apelarea unei funcţii parametrii formali sunt înlocuiţi, poziţie cu poziţie, prin valorile parametrilor actuali iar rezultatul este furnizat exclusiv prin numele funcţiei. La proceduri, comunicaţia cu ajutorul parametrilor între programul apelant şi procedură are dublu sens, fiind posibilă, în plus, returnarea prin parametri a rezultatelor.

Pentru procedură, datele de intrare se transmit prin aşa-numiţii **parametri constantă (valoare)**, prin valoarea acestora, iar rezultatele se transmit prin intermediul **parametrilor variabilă,** care trebuie precedaţi în lista parametrilor formali de cuvântul-cheie **var.** Valoarea parametrilor variabilă poate fi modificată prin instrucţiunile procedurii, modificările fiind apoi vizibile în cadrul programului apelant. Parametrii variabilă pot fi folosiţi de către procedură şi pentru introducerea de date, ca în exemplul următor.

Exemplul 4. Următoarea procedură calculează media unui eşantion de n+1 date (numere), cunoscând :

— n;
— media primelor n numere ;
— cel de-al (n+1)-lea număr.

De asemenea, înlocuieşte valoarea mediei date cu valoarea mediei recalculate şi reactualizează dimensiunea eşantionului de la valoarea n la valoarea n + 1.

```
procedure NouaMedie (x:real; var n:integer; var medie:real);
begin
    medie:=(medie*n+x) / (n+1);
    n:=n+1
end;
```

Procedura poate fi folosită într-un program astfel :

```
NrDate:=3;
NouaData:=4 .0;
Media:=2 .0;
NouaMedie(NouaData, NrDate, Media);
write ('Media celor', NrDate:3,' date este : ',Media:6:2);
```

Rezultatul va fi :

Media celor 4 date este : 2.50

Exemplul 5. Procedură pentru transformarea unei distanţe exprimate în metri într-o distanţă exprimată în unităţile de măsură anglo-saxone (feet, inches).

```
procedure FeetInches (metri:real; var feet, inches:real);
const f=3.2808; {feet/metri}
var temp:real;
begin
    temp:=metri*feet;
    feet:=trunc(temp);
    inches:=(temp-feet)*12
end.
```

Instrucţiunea de apelare poate fi, de exemplu :

FeetInches(1933,f,i);

Exemplul 6.

Următorul subprogram este o procedură fără parametri.

```
procedure Antet;
```

```
begin
    writeln('*************************************************');
    writeln('* Nr. crt.  *  Numele  *   Vârsta  *');
    writeln ('*************************************************')
end;
```

Instrucţiunea de apelare este pur şi simplu :
Antet ;
şi are ca efect tipărirea capului de tabel evidenţiat în textul de mai sus.

Diagramele de sintaxă asociate noţiunii de procedură sunt prezentate în figurile 10.9—10.11, astfel :

— în figura 10.9 diagrama de sintaxă a listei de parametri formali ;
— în figura 10.10 secţiunea de parametri constantă ;
— în figura 10.11 secţiunea de parametri variabilă.

Observaţii :

1. Lista parametrilor formali poate fi vidă, atunci când nu există schimb de informaţii cu restul programului sau când acest schimb se execută doar prin intermediul variabilelor globale.

2. Parametrul formal constantă (valoare) este tratat în cadrul subprogramului ca o variabilă locală. La apelarea procedurii i se alocă spaţiu şi este iniţializat cu valoarea parametrului actual corespunzător apoi subprogramul poate modifica valoarea acestuia fără a modifica parametrul actual corespunzător. Deci el poate doar să transmită date de la programul principal la procedură.

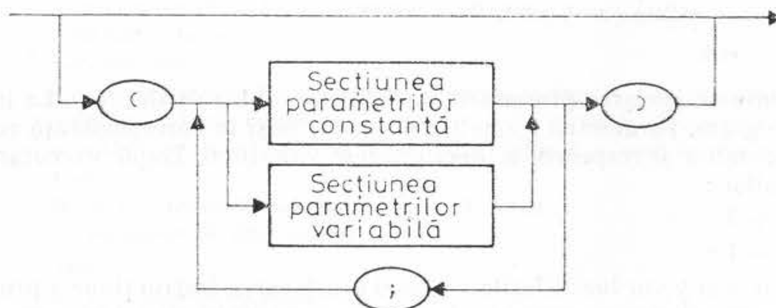


Fig. 10.9.

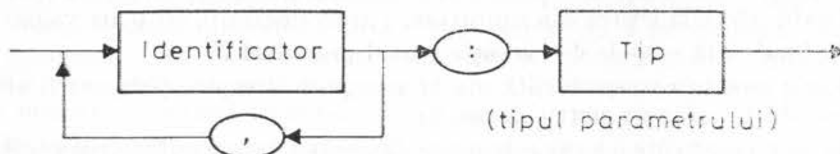


Fig. 10.10.

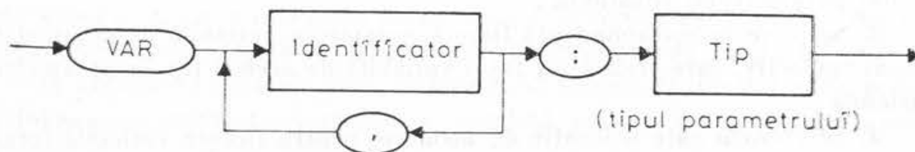


Fig. 10.11.

3. În replică, modificările asupra parametrilor formali variabilă se transmit parametrilor actuali.

4. Utilizarea variabilelor locale se recomandă pentru memorarea și utilizarea unor valori ce nu se transmit în afara subprogramului. Ca avantaje ale folosirii variabilelor locale, se pot menționa : siguranța (variabilele locale nu pot fi afectate de alte subprograme), claritatea, utilizarea eficientă a memoriei (variabilele locale au „viață scurtă“, cele globale au „viață lungă“).

Exemplul următor ilustrează cea de-a doua observație.

Exemplul 7.

```
program exerciţiu ;
var a, b:real;
procedure tipăreşte (x:real; var y:real);
  begin
    x:=x+1;
    y:=y+1;
    writeln ('x= ', x:3:1, 'y=',y:3:1);
  end;
begin
  a:=0;
  b:=0;
  tipăreşte (a, b);
  writeln ('a=', a:3:1, 'b=', b:3:1)
end.
```

Înainte de apelarea procedurii, variabilele a și b au valorile 0. La intrarea în subprogram, parametrii formali x și y sunt puși în corespondență cu parametrii actuali a și respectiv b, deci primesc valorile 0. După executarea instrucțiunilor :

x : = x + 1 ;

y : = y + 1 ;

variabilele x și y vor lua valorile 1 și deci următoarea instrucțiune a procedurii va determina tipărirea acestor valori. Deoarece numai y și deci, respectiv, b sunt parametrii variabilă (în lista de parametri formali numai y este precedat de cuvîntul cheie **var**), la ieșirea din subrutină numai b va păstra valoarea modificată. Deci, la ieșirea din subrutină, a are valoarea 0, iar b are valoarea 1.

În final, iată **etapele de execuție a unei proceduri :**

1. Se pun în corespondență, de la stânga la dreapta, parametrii actuali ai procedurii cu parametrii formali ;

2. Se creează câte o locație de memorie pentru fiecare parametru constantă (valoare) și i se atribuie acesteia valoarea parametrului actual asociat care trebuie să fie o expresie corespunzătoare, din punctul de vedere al atribuirii, tipului parametrului constantă ;

3. Se pune în corespondență fiecare parametru variabilă cu parametrul actual respectiv, care trebuie să fie o variabilă de același tip cu parametrul variabilă ;

4. Se creează câte o locație de memorie pentru fiecare variabilă locală a procedurii. Valoarea inițială a acesteia este nedefinită ;

5. Se execută instrucţiunile procedurii. În această etapă parametrii constantă (valoare) se comportă la fel ca oricare variabilă locală, în schimb orice referire la parametrii variabilă constituie o referire la parametrii actuali pe care aceştia îi reprezintă. În particular, atribuirea unei valori unui parametru variabilă modifică în mod efectiv valoarea parametrului actual ;

6. La sfîrşitul procedurii, toate locaţiile create sunt eliberate.

10.3. PROCEDURI ŞI DATE DE TIP STRUCTURAT

Atunci când subprogramele şi datele de tip structurat sunt folosite împreună, ele constituie un instrument deosebit de util pentru organizarea programelor. Elementele componente ale datelor de tip structurat se pot folosi ca parametri în subprograme la fel ca şi variabilele simple.

Exemplul 8. Următoarea procedură are ca parametru actual o variabilă de tip integer şi realizează incrementarea cu o unitate a valorii acesteia.

```
procedure increment (var contor:integer);
  begin
    contor:=contor+1
  end ;
```

Dacă dorim să scriem un program care să contorizeze numărul de apariţii ale fiecărei cifre şi numărul de apariţii ale celorlalte caractere într-un text dat, va fi necesar să folosim variabilele:

```
FCifra: array ['0'..'9'] of integer;
FAlteCar: integer;
Caracter:char;
```

şi să iniţializăm FAlteCar, precum şi fiecare componentă a tabloului FCifre cu valoarea 0. După memorarea unui caracter în variabila Caracter, frecvenţa de apariţie a acestuia în text va putea fi reactualizată folosind următoarea instrucţiune :

```
if ('0' <= Caracter) and (Caracter <='9') then
   increment (FCifra[Caracter])
else
   increment (FAlteCar) ;
```

În momentul execuţiei instrucţiunii increment (FCifra[Caracter]), parametrul variabilă Contor reprezintă variabila FCifra[Caracter], adică o singură componentă a tabloului FCifra. De exemplu, dacă în variabila Caracter este memorată valoarea '7', Contor reprezintă FCifra['7'], adică numărul de apariţii (frecvenţă) în text ale caracterului '7'. Prin executarea procedurii va fi modificată numai valoarea acestei componente a tabloului.

Majoritatea programelor care folosesc date de tip structurat conţin subprograme de prelucrare a acestora. De cele mai multe ori se preferă ca toate referirile la o structură să fie încorporate în subprograme, ceea ce conferă întregului program o mai mare independenţă în raport cu detaliile structurii respective şi prin urmare îl face mai uşor de înţeles şi de modificat.

În următorul exemplu sunt incluse două proceduri (CitMat şi ScrieMat) ce folosesc ca parametru o structură completă, nu doar o componentă a acesteia.

Exemplul 9.

```
program ProdMat;
type mat=array[1..10, 1..10] of real;
var A,B,C:mat;
    i, j, k, l, n, m:integer;
procedure CitMat (var A:mat;n, m:integer);
    begin
                for i: = 1 to n do
                    for j:=1 to m do
                        read (A[i, j])
    end; {citire matrice}
procedure ScrieMat (var A:mat;n,m:integer);
    begin
                for i:=1 to n do
                    begin
                        for j:=1 to m do
                            write(A[i,j]:6:2);
                        writeln
                    end
    end; (scriere matrice}
begin {ProdMat}
    writeln('Introduceţi valori pentru n, l, m:');
    read(n, l, m);
    writeln('Introduceţi valorile matricii A:');
    CitMat (A, n, l);
    writeln('Matricea A este');
    ScrieMat (A, n, l);
    writeln('Introduceţi valorile matricii B:');
    CitMat(B, l, m);
    writeln('Matricea B este');
    ScrieMat(B, l, m);
    for i: = 1 to n do
        for j:=1 to m do
            begin
                C[i, j]:=0;
                for k:= 1 to l do
                    C[i, j]:=C[i, j]+A[i, k]*B[k, j]
            end;
    writeln('Matricea produs este');
    ScrieMat(C, n, m);
    readln;
    readln
end.
```

• Ca **regulă generală**, dacă se doreşte ca o procedură să nu modifice valoarea unui parametru este bine ca acesta să fie declarat ca **parametru valoare**. în acest mod protejându-se parametrul actual faţă de orice modificare operată de către procedură asupra parametrului formal. Totuşi, deoarece pentru parametrii valoare se alocă spaţiu de memorie la fiecare apelare a procedurii, în cazul în care parametrul respectiv este o **structură** de dimensiune considerabilă (de exemplu un tablou) această variantă consumă atât spaţiu de memorie

cât și timp de execuție. În acest caz se preferă ca parametrul să fie declarat ca **parametru variabilă** și să se insereze în procedură un **comentariu** care să precizeze că acesteia nu îi este permis să modifice valoarea parametrului respectiv.

10.4. IERARHIZAREA PE NIVELURI A PROCEDURILOR ȘI FUNCȚIILOR. DOMENII DE VALABILITATE ALE IDENTIFICATORILOR

Declararea unui identificator în interiorul unui subprogram nu are efect în exteriorul acestuia și de aceea se spune că este vorba despre un **identificator local.** Pe de altă parte, un *identificator declarat* în programul principal poate fi folosit oriunde și se numește **identificator global.**

Pentru clarificarea acestor noțiuni se vor prezenta în continuare **regulile ce stabilesc domeniul de valabilitate al identificatorilor (scope rules)**, adică regulile care precizează ce identificatori pot fi folosiți și unde anume. În acest scop se va utiliza un program în care sunt încadrate în câte un chenar părțile componente ale programului principal și ale subprogramelor aferente. Aceste componente se numesc **blocuri.** Blocurile pot fi **„imbricate"**, adică un bloc poate fi cuprins în întregime în cadrul altui bloc și ca o consecință a sintaxei limbajului Pascal, acesta este singurul mod în care ele se pot suprapune.

În exemplul prezentat apare un singur nivel de imbricare ; în programele cu dimensiuni mari pot exista însă mai multe niveluri de imbricare. În general, va exista câte un bloc pentru fiecare funcție și procedură și un bloc pentru întregul program, iar parametrii formali ai funcției sau ai procedurii vor fi incluși în blocul corespunzător.

Fiecare bloc va fi desemnat prin identificatorul funcției, procedurii sau al programului.

În exemplul prezentat în figura 10.12 există un bloc exterior, Afișare și două blocuri interioare, CiteșteMesaj și ScrieMesaj.

● **Regula nr. 1 :** *în cadrul unui bloc, un identificator poate fi declarat o singură dată.*

Este evident că dacă în același bloc există două declarații ale identificatorului X orice apariție a acestuia în cadrul unei instrucțiuni va avea un înțeles ambiguu. Totuși, un același identificator poate fi declarat în blocuri diferite.

În exemplul prezentat, L este definit drept constantă în blocul Afișare și este declarat ca parametru valoare în blocul ScrieMesaj, acestea constituind două entități distincte. Același lucru se întâmplă și cu identificatorul j care, deși utilizat în ambele blocuri cu declarații de același tip, reprezintă două variabile distincte.

În cazul în care se folosește un identificator pentru care există mai multe declarații se aplică următoarea regulă :

● **Regula nr. 2 :** *pentru a stabili care este tipul unui identificator* X care apare într-o instrucțiune *se caută cel mai mic bloc ce include atât instrucțiunea, cât și declararea lui* X. Cea din urmă stabilește tipul identificatorului respectiv.

Luând în discuție identificatorul L, din exemplul prezentat :

— în blocul Afișare, L este folosit de instrucțiunea ScrieMesaj (51, L). Aplicând Regula nr. 2, constatăm că cel mai mic bloc care include atât această instrucțiune cât și declararea lui L este blocul Afișare și prin urmare L este constanta 80 ;

Exemplul 10

```
program Afişare;
 const L=80;
 type număr=1..L;
 var Mesaj:array [număr] of char;

 procedure CiteşteMesaj;
   var j:număr;
   begin
   for j:=1 to L do read(Mesaj[j]);
   readln
   end {CiteşteMesaj};

 procedure ScrieMesaj (L,U : număr);
   var j:număr;
   begin
   for j:=L to U do write(Mesaj[j]);
    writeln
   end {ScrieMesaj};

 begin {Afişare}
    CiteşteMesaj;
    ScrieMesaj(1,25);
    ScrieMesaj(26,50);
    ScrieMesaj(51,L)
 end {Afişare}.
```

Fig. 10.12.

— în blocul CiteşteMesaj, L apare în instrucţiunea **for** dar în acest bloc nu există declararea lui L şi prin urmare şi aici se aplică definirea lui L drept constantă deoarece Afişare este cel mai mic bloc care include atât instrucţiunea **for** cât şi declararea lui L;

— în blocul ScrieMesaj, L apare în instrucţiunea **for** dar în acest caz există o declaraţie a lui L ca parametru valoare al procedurii şi prin urmare L va lua valoarea parametrului actual corespunzător, adică 1, 26 sau 51.

În ceea ce priveşte identificatorul j, acesta este folosit ca variabilă contor în blocurile CiteşteMesaj şi ScrieMesaj, în fiecare dintre acestea existând declararea lui j. Prin urmare, se va aplica în fiecare caz declaraţia locală corespunzătoare.

Observaţie: Dacă, aplicând Regula nr. 2, nu reuşim să găsim un bloc care să cuprindă atât instrucţiunea, cât şi declaraţia identificatorului X, rezultă că programul conţine o eroare:

— fie a fost omisă declaraţia;

— fie declaraţia a fost plasată într-un bloc necorespunzător.

• **Regula nr. 3** : *un identificator nu poate fi folosit în afara blocului în care a fost declarat.*

De exemplu, nu e permisă folosirea identificatorului U în blocul Afişare sau în blocul CiteşteMesaj. El poate fi utilizat doar în cadrul blocului ScrieMesaj.

Cele trei reguli se aplică asupra tuturor identificatorilor : constante, tipuri, variabile, parametri formali, funcţii, proceduri. Orice încălcare a acestor reguli constituie o eroare, fiind semnalată ca atare de către compilator. În cazul în care dispunem de un program scris pe mai multe niveluri, ca în figura 10.13.

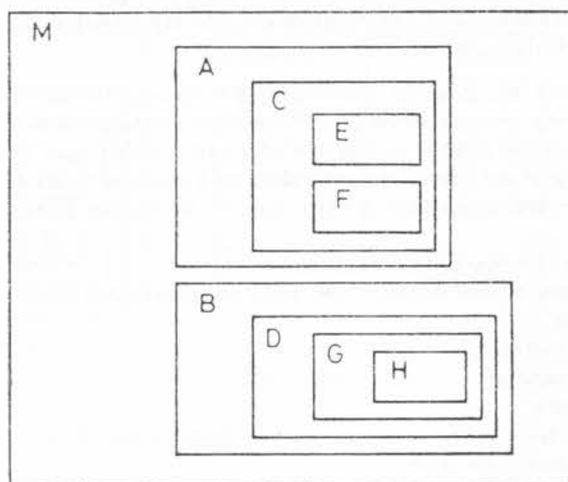


Fig. 10.13.

se consideră următoarea ierarhizare a blocurilor :

Bloc	Nivel
M	0
A, B	1
C, D	2
E, F, G	3
H	4

Domeniile de valabilitate ale identificatorilor vor fi :

Identificatori declaraţi în blocul :	Sunt accesibili în blocul :
M (global)	M, A, B, C, D, E, F, G, H
A	A, C, E, F
B	B, D, G, H
C	C, E, F
D	D, G, H
E	E
F	F
G	G, H
H	H

O variabilă declarată, de exemplu, în blocurile B şi G şi apelată în blocul H, va avea tipul precizat în declaraţia conţinută de blocul G.

10.5. UTILIZAREA FUNCȚIILOR ȘI PROCEDURILOR CA PARAMETRI AI SUBPROGRAMELOR

Limbajul de programare PASCAL asigură posibilitatea apelării unui subprogram din interiorul altui subprogram. Acest lucru pare foarte firesc și, într-adevăr, este de neimaginat o structură modulară, ierarhizată, a programelor în absența acestei facilități. Mai mult decît atît, limbajul PASCAL permite ca, la momente diferite de timp să fie posibilă utilizarea de către un subprogram a unor funcții sau proceduri distincte, adecvate necesităților de prelucrare ale momentului respectiv. Această posibilitate este asigurată de mecanismul transmiterii unor parametri de tip funcție sau procedură, evidențiat în exemplul următor.

Exemplul 11. Program pentru calculul aproximativ al minimului unei funcții F definite pe un interval [A, B] specificat. Determinarea se face prin evaluări succesive ale funcției în punctele (A+i*(B—A)/N), unde N reprezintă numărul de divizări ale intervalului menționat iar i primește valori de la 0 la N. Precizia determinării minimului funcției depinde de finețea divizării intervalului.

```
...
type TipF=function (x:real):real;
procedure MinimFuncție(F:TipF;A,B:real;N:integer;var Min:real);
   var
     i:integer;
     pas:real;
   begin
     Min:=F(A);
     pas:=(B—A)/N;
     for i:=0 to N do
        if Min > F(pas*i+A) then Min:=F(pas*i+A)
   end;
```

În exemplul anterior se evidențiază în lista parametrilor formali așa-numita **secțiune a parametrilor funcționali :**
F : TipF ;
unde s-a definit anterior
type TipF = **function** (x : real) : real ;

În secțiunea parametrilor funcționali (procedurali), numele variabilelor ce figurează ca parametri pentru funcțiile sau producerile respective pot fi oricare, existând libertate deplină în alegerea lor. Ceea ce trebuie respectat cu strictețe la apelarea unui subprogram cu parametri funcționali sau procedurali este, pe de o parte, *tipul funcției* (dacă se folosește o funcție) iar pe de altă parte, *numărul și tipul parametrilor proprii* funcției sau procedurii ce figurează ca parametru actual. Astfel, considerând exemplul de mai sus, putem apela procedura MinimFuncție astfel :

MinimFuncție (G, 0, 10, 100, MinS) ;
unde funcția G este definită ca mai jos :
function G (x : real) : real ;
  begin
    G : = sqr(x) — 4*x — 12
  end ;

Desigur, în cazul prezentat, ca parametru al producerii MinimFuncție poate fi folosită orice alta funcție care este de tip real cu un singur parametru de tip real, așa cum impune declarația tipului TipF.

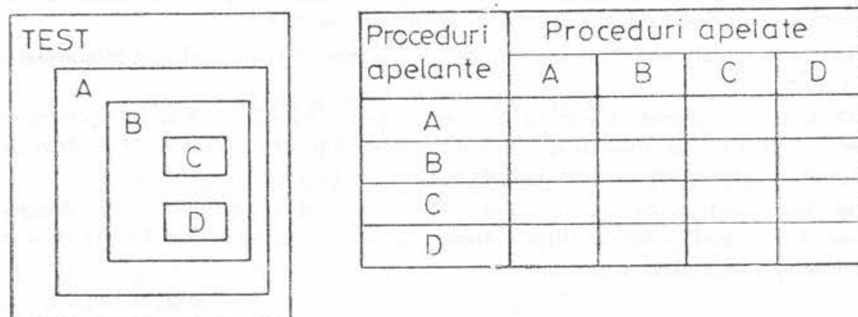
Observaţie : Exemplul anterior şi-a propus prezentarea modului de utilizare a parametrilor de tip subprogram şi nu a algoritmului de calcul al minimului unei funcţii, pentru acesta existând soluţii mai eficiente.

EXERCIŢII

1. Ce diferenţă există între o variabilă globală şi o variabilă locală ?

2. Care sunt regulile de vizibilitate a numelor ?

3. În următorul tabel, marcaţi cu steluţă procedura din dreapta ce poate fi referită de procedura corespunzătoare din stânga, ştiind că programul are structura :



Proceduri apelante	Proceduri apelate			
	A	B	C	D
A				
B				
C				
D				

Fig. 10.14.

4. Ce deosebiri există între funcţii şi proceduri ?

5. Verificaţi corectitudinea următoarei declaraţii de funcţie :

```
function F3(var a:real):boolean;
   begin
      F3:=a+5;
   end;
```

6. Considerând că programul principal conţine următoarele declaraţii :

```
var
x,y:real;
m:integer;
next:char;
```

procedure Test (a, b:integer; **var** c,d:real ; **var** e:char);
care dintre apelurile următoare sunt incorecte ? Motivaţi răspunsul.

a. Test (m, MaxInt, y, x, next);
b. Test (m, 10, 35, y, 'E');
c. Test (m, 10, x, y, next);
d. Test (m, 19, x, y) ;
e. Test (m, 3.0, x, y, next);
f. Test (30, 10, m, x, next);

7. Scrieţi o funcţie care să furnizeze ca rezultat puterea a patra a unei valori de tip real. Scrieţi apoi o instrucţiune prin care să se calculeze $(A + B)^4$, unde A şi B sunt două valori reale.

8. Să se scrie o funcţie care să determine cel mai mare divizor comun pentru două numere întregi precizate.

9. Scrieţi o funcţie care să aibă ca rezultat valoarea minimă a unui tablou de numere.

10. Să se scrie o funcţie care să determine cel mai mic multiplu comun pentru două numere întregi precizate.

11. Scrieţi o funcţie care să aibă ca rezultat suma cifrelor ce formează un număr întreg.

12. Să se scrie o funcţie care să calculeze suma elementelor unui tablou, având ca parametri tabloul şi numărul său de elemente.

13. Să se scrie o procedură care „roteşte" cu 90° o matrice pătrată.

14. Să se scrie o funcţie CIFRA(n, m) care are ca rezultat valoarea celei de-a m – a cifre de la dreapta spre stânga a numărului n scris în sistemul zecimal. De exemplu : CIFRA (7283,3) are valoarea 2.

15. Să se definească o procedură care inserează într-o listă dată cu N elemente ordonate crescător un nou element, astfel ca lista obţinută să fie ordonată crescător. Se va utiliza apoi această procedură pentru a comasa două liste A şi B, având P respectiv Q elemente ordonate crescător într-o singură listă A, ordonată de asemenea crescător.

16. Să se înmulţească două matrici utilizând o funcţie pentru calculul produsului scalar a doi vectori.

17. Definiţi o procedură care să stabilească dacă o valoare dată se află printre cele N elemente ale unui şir dat. Folosiţi apoi această procedură pentru a crea un şir de elemente distincte pe baza unor valori citite de la tastatură.

18. Într-o matrice dată A cu L linii şi C coloane, să se permute circular dreapta fiecare linie I cu I poziţii. Se va utiliza o procedură care permută circular dreapta cu o poziţie componentele unui vector.

CAPITOLUL 11

SUBPROGRAME RECURSIVE

11.1. RECURSIVITATE. ALGORITMI RECURSIVI

În general, prin recursivitate se înțelege proprietatea intrinsecă a unei entități (proces, obiect, fenomen etc.) de a putea fi descrisă, prelucrată, analizată etc. pe baza unor entități de același tip. Mai simplu spus, în definirea sau prelucrarea unei entități recursive se fac referiri sau apeluri la însăși entitatea respectivă. Pentru a exemplifica noțiunea de recursivitate se consideră următoarele situații :

Exemplul 1. Definirea numerelor naturale.

— 1 este număr natural ;

— succesorul unui număr natural este un număr natural.

Se presupune cunoscută definiția succesorului unui număr : acel număr obținut din numărul dat prin adăugarea unei unități.

Exemplul 2. Algoritm de calcul pentru factorialul unui număr N.

— Dacă N=0 atunci N ! =1;

— Dacă N > 0 atunci N!=N*(N—1)!

Altfel spus, factorialul unui număr N > 0 se obține prin înmulțirea numărului cu factorialul predecesorului.

Algoritmii recursivi sunt acei algoritmi ce conțin apeluri la ei înșiși. Alături de exemplul precedent, se poate adăuga algoritmul lui Euclid pentru determinarea celui mai mare divizor comun a două numere. Pentru ilustrarea acestuia se reamintește, pe un caz concret, modalitatea de lucru.

Exemplul 3. Algoritmul lui Euclid pentru determinarea celui mai mare divizor comun pentru două numere date.

Fie 27 și 15 cele două numere considerate. Se efectuează o secvență de împărțiri succesive, împărțitorul și restul asociate unei operații de împărțire reprezentând deîmpărțitul și respectiv împărțitorul pentru următoarea operație.

Nr. operație	Operație	Cât	Rest
1	27:15	1	12
2	15:12	1	3
3	12:3	4	0

În momentul în care restul obținut este 0, ultima valoare folosită ca împărțitor reprezintă cel mai mare divizor comun al celor două numere considerate inițial, adică, în cazul de față, cel mai mare divizor comun al numerelor 27 și 15 este 3.

Algoritmul lui Euclid poate fi exprimat, într-o formă recursivă, astfel: „Fie A și B două numere naturale. Cel mai mare divizor comun al celor două numere, notat c.m.m.d.c. (A, B), se calculează în felul următor:
— dacă B = 0 atunci c.m.m.d.c. (A, B) = A;
— altfel, c.m.m.d.c. (A, B) = c.m.m.d.c. (B, rest (A, B))."
S-a notat prin rest (A, B) restul împărțirii lui A la B.

Din exemplele anterior prezentate se poate sesiza o puternică legătură între recursivitate și iterație. Într-adevăr, în cazul algoritmilor, execuția recursivă presupune repetarea apelului la însuși algoritmul considerat, adică iterarea unor operații de calcul. Pentru ca această iterație să nu devină infinită este necesar ca în corpul algoritmului să se prevadă cel puțin o testare a unei condiții de oprire (în ultimul exemplu, apelul recursiv se oprește când B = 0).

Se poate demonstra că, în general, orice recursivitate poate fi transformată în iterație propriu-zisă (cu alte cuvinte, „autoapelul" poate fi eliminat).

11.2. PROCEDURI ȘI FUNCȚII RECURSIVE

Evidențiată la nivelul algoritmilor, recursivitatea se translatează în mod firesc la nivelul implementărilor acestora în limbajele de programare. Instrumentele cele mai adecvate pentru aceste implementări sunt, desigur, procedurile și funcțiile, generic denumite „subprograme". Există două tipuri de subprograme recursive:
— direct recursive;
— indirect recursive.

Mai exact, un subprogram S, în corpul căruia apar apeluri la S (la el însuși) se numește subprogram **direct recursiv** iar un subprogram P, pentru care există un subprogram Q, astfel încât P face apeluri la Q iar Q conține apeluri la P se numește subprogram **indirect recursiv.** În acest ultim caz, subprogramele P și Q se mai numesc și **mutual recursive.**

În figura de mai jos sunt ilustrate, schematic, (urmând a fi detaliate mai târziu), structurile unor proceduri recursive.

● *Procedură direct recursivă*

```
procedure S;
  begin
    ...
    S; {apel la procedura S}
    ...
  end;
```

● *Proceduri mutual recursive*

```
procedure P;
  begin
    ...
    Q; {apel la procedura Q}
    ...
  end;
procedure Q;
  ...
  begin
    ...
    P; {apelul procedurii P}
    ...
  end;
```

În PASCAL, transcrierea recursivă a calculului factorialului poate fi următoarea :

```
function Factorial (N : integer) : integer ;
begin
  if N = 0 then Factorial : = 1
  else Factorial : = N*Factorial (N – 1)
end ;
```

De remarcat că, așa cum s-a precizat la sfârșitul paragrafului precedent, și funcția recursivă Factorial conține o decizie (**if** N = 0) care, la un moment dat, în cursul execuției funcției, asigură oprirea apelului recursiv, forțând ieșirea din subprogram.

Reamintind că într-unul dintre capitolele precedente a apărut și forma nerecursivă a funcției Factorial, se prezintă în continuare cele două variante (iterativă și recursivă) pentru algoritmul lui Euclid (descris în exemplul 3).

• *Varianta iterativă* :

```
function cmmdc (a, b : integer) : integer ;
  var t : integer ;
  begin
    while b <> 0 do
      begin
        t : = b ;
        b : = a mod b ;
        a : = t
      end ;
    cmmdc : = a
  end ;
```

• *Varianta recursivă :*

```
function cmmdc (a, b : integer) : integer ;
  begin
    if b = 0 then cmmdc : = a
    else cmmdc : = cmmdc (b, a mod b)
  end ;
```

Înainte de a intra în detalii asupra modului concret de execuție a unui subprogram recursiv, trebuie amintit că, în majoritatea cazurilor, soluția iterativă de scriere este preferabilă celei recursive. Subprogramele recursive sunt mult mai concise, însă mult mai greu de controlat și depanat (riscul unor greșeli de concepție este mare !). De asemenea, în cazul subprogramelor recursive este nevoie de un spațiu mai mare de memorie disponibilă iar timpul de execuție este mai ridicat.

Pe de altă parte, există cazuri în care exprimarea nerecursivă a unui algoritm este mai dificilă, mai puțin elegantă sau afectează eficiența acestui algoritm.

De regulă, se folosesc algoritmi recursivi (și deci și subprograme recursive) în cazurile în care calculele aferente sunt descrise în formă recursivă.

Practic recursivitatea este frecvent folosită în procedurile de prelucrare a structurilor de date definite recursiv (arbori, liste etc.).

Pentru a înţelege în detaliu modul în care se execută un subprogram recursiv trebuie mai întîi cunoscut ce se întîmplă la apelul unei funcţii sau proceduri oarecare (nerecursive). Implementarea apelurilor de proceduri şi funcţii se face prin intermediul unei structuri de date interne, numită **stivă**. Prin stivă se înţelege o *zonă rezervată de memorie, în care se poate efectua salvarea temporară a valorilor unor variabile.*

Deşi există mai multe modalităţi de lucru cu stiva, în PASCAL trebuie avut în vedere mecanismul LIFO (Last-In-First-Out, adică „ultimul intrat-primul ieşit") ce guvernează introducerea şi extragerea informaţiilor din stivă. Altfel spus, extragerea din stivă se face în ordinea inversă a introducerii de date, fiecare introducere (salvare) de informaţie în stivă mărind dimensiunea stivei şi fiecare extragere (restaurare) reducându-i acesteia dimensiunea. Acest lucru este ilustrat în figura 11.1. Stiva se completează în ordinea descrescătoare a adreselor de memorie. Sensul de creştere a adreselor de memorie corespunde „coborârii" spre marginea inferioară a desenului iar completarea stivei se face, aşadar, prin „urcarea" spre adrese inferioare.

În general, în momentul în care un program P apelează un subprogram S are loc o depunere (salvare) în stivă a unei adrese de revenire şi a contextului programului P.

Mai precis, **adresa de revenire** serveşte *controlului execuţiei generale a programului,* ea indicând instrucţiunea din programul P ce urmează a fi executată la sfârşitul execuţiei subprogramului S.

Prin **contextul** unui subprogram înţelegem *totalitatea variabilelor sale locale.* Pe lângă operaţiile de salvare în stivă, la apelul unui subprogram se alocă spaţiu pentru variabilele locale asociate subprogramului. Se consideră

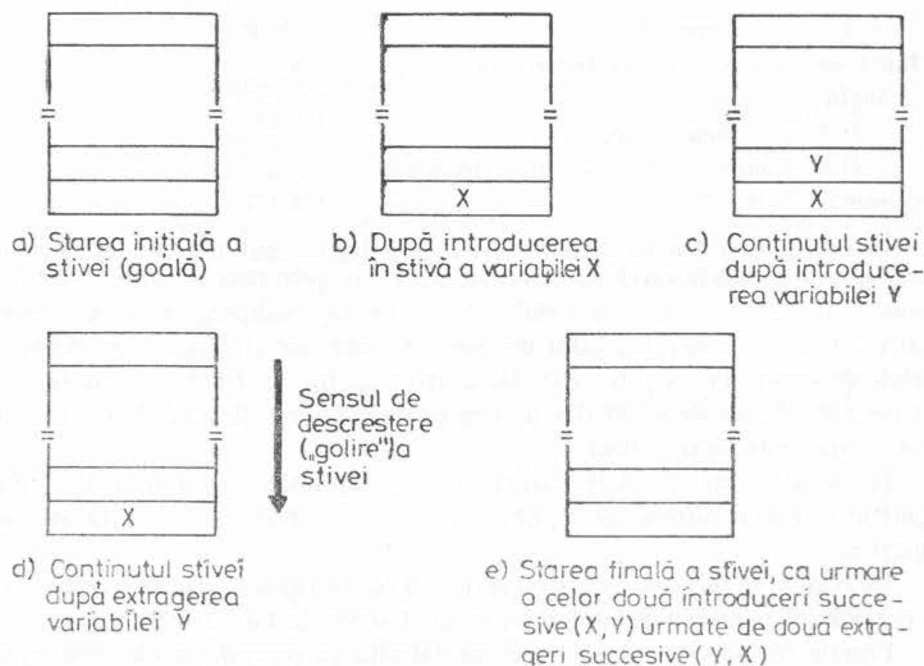


Fig. 11.1.

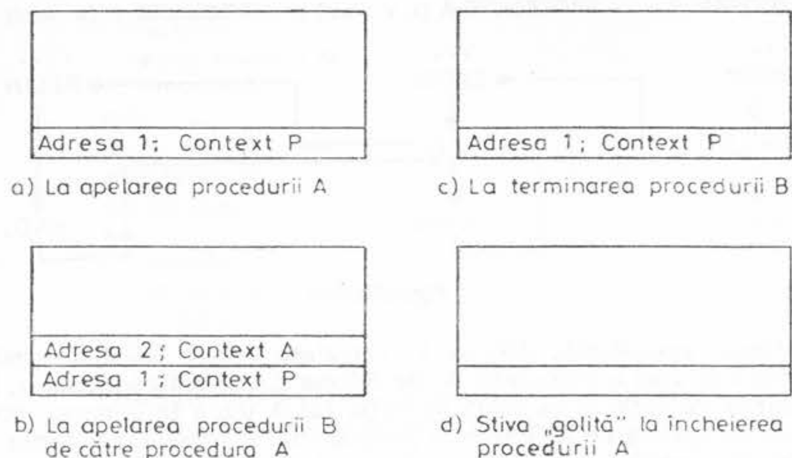


Fig. 11.2.

un program P ce utilizează o procedură A (primul nivel de subordonare) care la rândul ei folosește o procedură proprie B (al doilea nivel de subordonare — privind dinspre vârful ierarhiei. Atenție! P nu poate apela direct procedura B). Structura unui astfel de program este redată în figura 11.3.

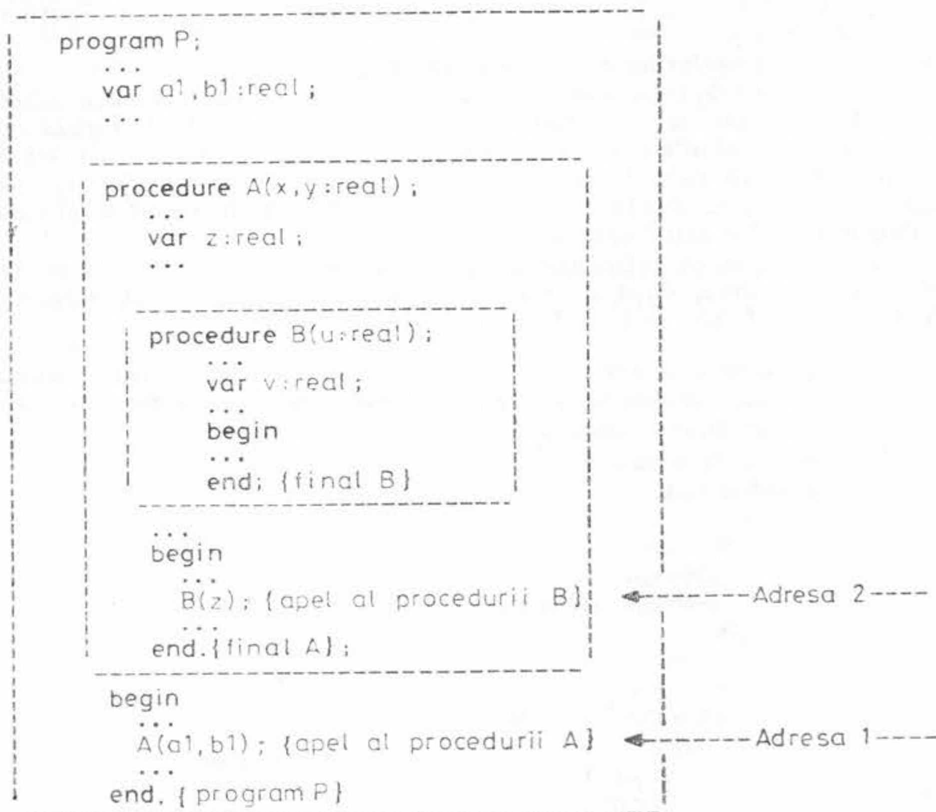


Fig. 11.3.

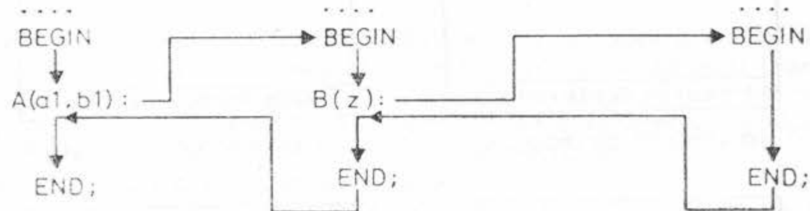


Fig. 11.4.

În figura precedentă, Adresa 1 reprezintă adresa imediat următoare instrucțiunii de apel a procedurii A, iar Adresa 2, în mod asemănător, indică instrucțiunea cu care se va relua execuția lui A după terminarea execuției procedurii B, apelată de A. Execuția programului P prezentat anterior poate fi descrisă din punctul de vedere al stivei ca în figura 11.2. Facem precizarea că desenele din figurile 11.1 și 11.2 au doar rolul de a sugera funcționarea mecanismului de stivă, ele nefiind riguroase din punctul de vedere al reprezentării informației în stivă.

Fluxul de control (ordinea de execuție a instrucțiunilor) în cazul rulării programului P este reprezentat(ă) în figura 11.4.

Urmărind săgețile, se poate identifica, pe ansamblu, ordinea de execuție a instrucțiunilor (nu adâncim detalierea la nivelul instrucțiunilor repetitive sau condiționale).

Și în cazul apelurilor recursive mecanismul este același. La fiecare apel recursiv al unui subprogram se salvează starea curentă a execuției sale (adresa instrucțiunii cu care se va continua execuția întreruptă și valorile variabilelor locale, inclusiv valorile parametrilor) și se alocă spațiu pentru un nou set de variabile. Cu toate că noile variabile locale au aceiași identificatori, orice referințe la acești identificatori se asociază ultimului set de variabile alocate, ambiguitatea fiind astfel exclusă.

La terminarea execuției unui subprogram apelat recursiv, se reia ultima execuție întreruptă, după ce s-a refăcut contextul său (salvat anterior). Pentru exemplificare, se consideră următoarea problemă.

Exemplul 4. Se cere scrierea unei proceduri care să afișeze în ordine inversă, pe mediul de ieșire (display), un șir de caractere primit de pe mediul de intrare (tastatură), șir de caractere terminat cu ' ' (blank).

• *Varianta nerecursivă:*

```
procedure back;
    var
        Car:char;
        i, j:integer;
        sir:array[1..80] of char;
    begin
        i:=0;
        read(Car);
        while Car < > ' ' do
          begin
            i:=i+1;
            sir[i]:=Car;
            read(Car)
```

```
        end;
      for j:=i downto 1 do
        write(sir[j])
    end;
```

• *Varianta recursivă :*

```
procedure back;
    var Car:char;
    begin
        read(Car);
        if Car < > ' ' then back;
        write(Car)
    end;
```

Observaţie : Ambele forme ale procedurii pot fi perfecţionate, introducând o limitare necesară a lungimii şirului (de exemplu la 80 de caractere).

Insistând asupra variantei recursive, se constată că unica variabilă locală a procedurii este Car. Adresa instrucţiunii write(Car), cea cu care trebuie să se continue execuţia procedurii după apelul recursiv, s-a notat cu a1. Dacă se presupune că de la tastatură se introduce şirul 'XYZ', atunci starea stivei şi acţiunile vizibile ale programului (ieşirea acestuia) sînt prezentate în figura 11.5.

Din punctul de vedere al fluxului de control, imaginea asociată este cea din figura 11.6. Şi din urmărirea săgeţilor se poate observa ordinea de execuţie a instrucţiunilor write :

write(' ')
write('Z')
write('Y')
write('X')

O procedură asemănătoare este cea pentru scrierea inversă a unui număr natural (de exemplu, pentru 1792 se va tipări 2971), dată în exemplul de mai jos :

Exemplul 5

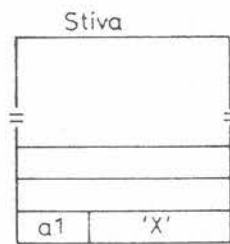
• *Varianta iterativă :*

```
procedure inver (n:integer);
    var x:integer;
    begin
        x:=n ;
        repeat
            write (x mod 10);
            x:=x div 10
        until x=0
        end;
```

• *Varianta recursivă :*

```
procedure inver (n:integer);
    begin
        if n > 0 then
            begin
                write (n mod 10);
                inver (n div 10)
            end;
    end;
```

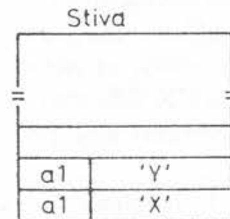
Observaţie : Mecanismul de salvare/restaurare a informaţiilor la apelurile de subprograme şi, în general, manipulările de date în care este implicată stiva, sunt transparente programatorului (pentru programele obişnuite el nu trebuie să definească stiva sau să scrie instrucţiuni pentru scriere/citire în/din stivă!



Efectul la display:
X

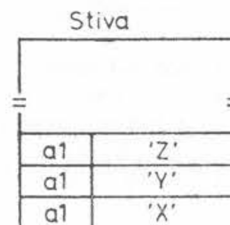
(ca urmare a instrucțiunii Read (car))

a) După apăsarea tastei X
(apel recursiv nr. 1)



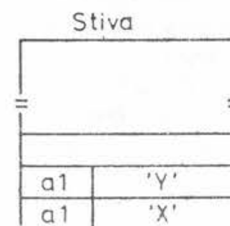
Efectul la display:
XY

b) După apăsarea tastei Y
(apel recursiv nr. 2)



Efectul la display:
XYZ

c) După apăsarea tastei Z
(apel recursiv nr. 3)

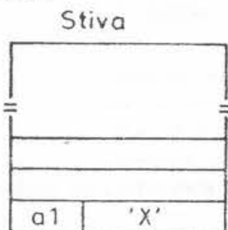


Efectul la display:
XYZ_
_

Fig. 11.5

Apoi:

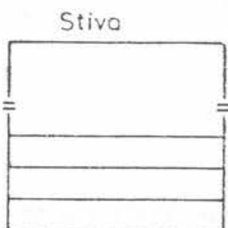
Stiva



Efectul la display:
XYZ_
_Z

Se încheie apelul de nivelul 2 extrăgîndu-se din stivă valoarea 'Y' pentru variabila CAR

Stiva



Efectul la display:
XYZ_
_ZY

Se finalizează apelul nr. 1, cu revenire la instrucțiunea WRITE (CAR) din corpul procedurii (apelul inițial)

În final stiva este "goală", iar efectul la display:

XYZ
_ZYX

Fig. 11.5 (continuare)

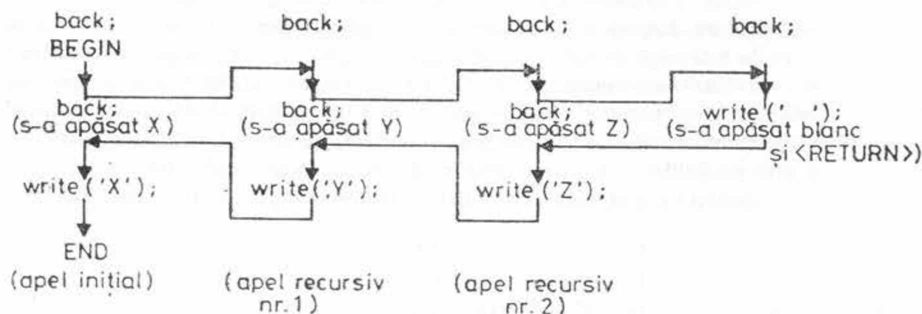


Fig. 11.6.

11.4. EXEMPLE DE PROGRAME RECURSIVE

Așa cum s-a precizat, subprogramele recursive reprezintă, în multe cazuri, soluții elegante pentru problemele de programare. Considerând expresia A la puterea N, unde A și N sunt întregi (N strict pozitiv), și asociind acesteia notația Putere (A, N) se poate da următoarea definiție recursivă:

$$\text{Putere}(A, N) = \begin{cases} A, & \text{dacă } N = 1 \\ A * \text{Putere}(A, N-1), & \text{pentru } N > 1 \end{cases}$$

Transcrierea în limbajul de programare PASCAL este imediată.

Exemplul 6.

```
type Pozitiv=1..MaxInt;
function Putere(A:integer; N:Pozitiv):integer;
    begin
        if N=1 then Putere:=A
        else Putere:=A*Putere(A,N—1)
    end;
```

Desigur, funcţia Putere poate fi scrisă utilizând o structură de control iterativă, ca în varianta de mai jos:

```
function Putere(A:integer;N:Pozitiv):integer;
    var i, p:integer;
    begin
        p:=1;
        for i:=1 to N do
            p:=p*A;
        Putere:=p
    end;
```

După cum s-a mai precizat, pentru multe probleme rezolvarea se poate găsi fie folosind recursivitatea, fie apelând la iteraţie. Deşi majoritatea programatorilor vor alege probabil soluţia din urmă, trebuie spus că anumite calcule sunt intrinsec recursive, implementarea lor în absenţa recursivităţii fiind deosebit de dificilă. Exemplul dat în continuare ilustrează afirmaţia de mai sus.

Exemplul 7. Problema labirintului.

Să ne imaginăm că labirintul are forma unei grădini englezeşti de formă rectangulară, împrejmuită de gard viu. De-a lungul perimetrului acesteia, gardul viu este întrerupt de una sau mai multe ieşiri. În interior, grădina este alcătuită din coridoare împrejmuite, la rândul lor, de garduri vii. Problema care se pune este aceea ca, pornind dintr-un punct interior grădinii, dintr-o poziţie iniţială aflată pe un coridor, să găsim calea ce conduce, parcurgând numai coridoare (fără a avea posibilitatea de a trece prin gardul viu !), la ieşirea din labirint.

Pentru început, este util să dăm o reprezentare labirintului (fig. 11.7).

```
+----------------+
I  HHHHHHHHHHHH  I
I  H..HHH....HH  I
I  HH...H.HHH.H  I
I  H..H.H.H...H  I
I  HHH....HHHHH  I
I  H...HHH....H  I
I  H.H.....HH.H  I
I  HHHHHHHHHH.H  I
+----------------+
```

Fig. 11.7.

Am figurat cu ajutorul caracterului 'H' gardul viu iar prin '.' spaţiul asociat coridoarelor. Se observă că, pentru reprezentarea acestei grădini-labirint, putem folosi un tablou bidimensional (ale cărui elemente sunt 'H' şi '.'). Suplimentar, vom folosi denumirea de parcelă, asociată unei arii elementare de pe suprafaţa grădinii. În reprezentarea iniţială a labirintului, unei parcele libere îi corespunde caracterul '.', iar o parcelă ocupată este marcată prin 'H'. Ulterior, pe măsură

ce se efectuează deplasări pe coridoarele labirintului, parcelele prin care am trecut deja se vor nota cu 'O', ele pierzând atributul de libere.

În rezolvarea problemei vom folosi pentru orientare mijloacele clasice: punctele cardinale. O primă schiță a soluției problemei labirintului este următoarea (notăm prin P poziția curentă pe coridoare, prin această poziție încercând să ne continuăm drumul spre ieșire, prin avansarea la o parcelă liberă din imediata vecinătate):

```
if parcela P se află pe conturul grădinii then am găsit ieșirea din labirintul
else
    begin
      încearcă să te îndrepți spre Est;
      if nu am găsit încă ieșirea then
           încearcă să te îndrepți spre Sud;
      if nu am găsit încă ieșirea then
           încearcă să te îndrepți spre Vest;
      if nu am găsit încă ieșirea then
           încearcă să te îndrepți spre Nord
end;
```

Soluția trebuie explicitată! Vom detalia înțelesul instrucțiunii „încearcă să te îndrepți spre Est", pentru celelalte trei puncte cardinale analogia fiind imediată:

```
{încearcă să te îndrepți spre Est}
if poziția vecină la Est lui P este o parcelă liberă (coridor)
then găsește o cale de la parcela vecină la Est lui P până la ieșire;
```

Evident, problema pusă: „găsește o cale de la parcela vecină la Est lui P până la ieșire" este echivalentă problemei originale: „găsește o cale de la P la ieșire"! Trebuie specificat că ordinea de investigare propusă (Est, apoi Sud, apoi Vest, apoi Nord) a fost aleasă arbitrar, problema rezolvându-se similar pentru orice altă combinație a celor patru puncte cardinale. Pentru rezolvare, vom folosi o procedură, GăseșteIeșire, având drept parametri coordonatele unei parcele și sarcina de a determina calea de ieșire din labirint, pornind din această parcelă. Procedura este de tip recursiv.

Programul de mai jos reprezintă implementarea soluției pentru un labirint de maximum 20 × 20 parcele.

```
program RezLabirint;
   const
      LimNord=1; LimSudMax=20; LimVest =1; LimEstMax=20;
      Coridor='.';
      Gard   ='H';
      Pas    ='O';
      Marcat ='*';
   type
      Latitudine=LimNord..LimSudMax;
      Longitudine=LimVest..LimEstMax;
      Parcele=char;
   var
      Labirint: array [Latitudine, Longitudine] of Parcele;
      StartLat: Latitudine;
      StartLong: Longitudine;
      LimSud, i:1..LimSudMax;
      LimEst, j:1..LimEstMax;
      GăsitIeșire: boolean;
```

```
procedure GăseşteIeşire (Lat: Latitudine; Long: Longitudine);
        begin
        if (Lat=LimNor) or (Lat=LimSud) or (Long=LimVest) or
           (Long=LimEst) then GăsitIeşire:=true
           else
              begin
                 Labirint [Lat, Long]:=Pas;
                 if Labirint[Lat, Long+1]=Coridor then
                    GăseşteIeşire(Lat, Long+1);
                 if not GăsitIeşire then
                    if Labirint[Lat+1, Long]=Coridor then
                       GăseşteIeşire (Lat+1, Long);
                 if not GăsitIeşire then
                    if Labirint [Lat, Long—1]=Coridor then
                       GăseşteIeşire (Lat, Long—1);
                 if not GăsitIeşire then
                    if Labirint [Lat—1, Long]=Coridor then
                       GăseşteIeşire (Lat—1, Long)
              end;
           if GăsitIeşire then Labirint [Lat, Long]:=Marcat
        end ;
begin {programul principal}
   {Secvenţa de citire a labirintului}
   write ('Dimensiune sud (maxim 20)=');
   readln (LimSud);
   write ('Dimensiune est (maxim 20)=');
   readln (LimEst);
   writeln;
   writeln ('Introduceţi pe linii structura labirintului');
   writeln ('Folosiţi caracterele:');
   writeln ('    H—gard');
   writeln ('    .—coridor');
   writeln;
   for i:=1 to LimSud do
      for j:=1 to LimEst do
         begin
            write ('Poziţia', i:2, ', ', j:2, ':');
            readln (Labirint [i, j])
         end;
   writeln;
   writeln ('Labirintul propus arată astfel:');
   writeln ;
   for i:=1 to LimSud do
      begin
      for j:=1 to LimEst do
         write (Labirint [i, j]);
      writeln
   end ;
   writeln;
   writeln ('Precizaţi poziţia iniţială');
```

```
  repeat
    write ('Linia   (1..', LimSud:2,')=');
    readln (StartLat);
    write ('Coloana (1..', LimEst:2,')=');
    readln (StartLong);
  until Labirint [StartLat, StartLong]='.';
  {Analiza căii de ieșire}
  GăsitIeșire:=false;
  GăseșteIeșire (StartLat, StartLong);
  {Secvența de afișare a labirintului}
  writeln ;
  writeln ('Labirintul rezolvat arată astfel:');
  writeln;
  for i:=1 to LimSud do
     begin
        for j:=1 to LimEst do
          write (Labirint [i, j]);
        writeln
     end;
  writeln;
  if not GăsitIeșire then
     writeln ('Nu există cale de ieșire din labirint');
  readln
end.
```

Prin instrucțiunea Labirint [Lat, Long]:=Pas ne asigurăm, că parcelele prin care am trecut sunt marcate, evitând posibila învârtire „în cerc", iar prin Labirint [Lat, Long]:=Marcat se evidențiază fiecare pas parcurs pe calea spre ieșire. În acest fel, în desenul final al labirintului apar trasate atât calea spre ieșire din labirint (marcată prin '*') cât și coridoarele parcurse inutil (marcate prin 'O') în timpul căutării. În figura 11.8 este reprezentată imaginea labirintului în urma determinării căii de ieșire pornind din parcela de coordonate (2,2):

```
+-----------------+
I   HHHHHHHHHHHH  I
I   H**HHHOOOOHH  I
I   HH***HOHHH.H  I
I   H..H*HOH...H  I
I   HHH**OOHHHHH  I
I   H..*HHH****H  I
I   H.H*****HH*H  I
I   HHHHHHHHHH*H  I
+-----------------+
```

Fig. 11.8.

Din exemplele prezentate se desprind două **principii generale referitoare la subprogramele recursive :**

1. *Orice subprogram recursiv trebuie să poată fi executat, în cel puțin o anume situație, fără a se autoapela.* În exemplul 6, această situație apare pentru N = 1, iar în problema labirintului (exemplul 7) atunci când parcela curentă este plasată pe conturul labirintului (reprezintă o ieșire). Mai mult, procedura GaseșteIexire din exemplul 7, nu se va invoca pe sine însăși nici în situația în care oricare dintre parcelele învecinate este fie acoperită de gard viu, fie a fost deja vizitată.

2. *Un subprogram recursiv se va autoapela într-un mod prin care se va tinde spre îndeplinirea condiției de execuție fără autoapelare* (de tipul celor evidențiate la 1).

Un caz particular al subprogramelor recursive îl constituie **subprogramele mutual recursive.** Utilizarea acestora presupune rezolvarea următoarei probleme legate de modalitatea concretă de declarare a acestora : în PASCAL există regula generală a declarării entităților (constante, variabile, funcții, proceduri etc.) înaintea folosirii lor (cu alte cuvinte, acestea trebuie cunoscute în momentul utilizării), regulă imposibil de respectat, prin mijloacele de care dispunem pînă în prezent, în cazul procedurilor (funcțiilor) indirect recursive. Evident, o declarare de tipul celei următoare este incompletă :

```
procedure P (lista de parametri formali);
  . . .
  begin {corp procedura P}
    . . .
    Q (lista de parametri actuali); {apelarea procedurii Q}
    . . .
  end;
procedure Q (lista parametri formali);
  . . .
  begin {corp procedura Q}
    . . .
    P (lista parametri actuali); {apelarea procedurii P}
    . . .
  end;
```

întrucât la nivelul instrucțiunii de apelare a procedurii Q, cuprinsă în procedura P, procedura Q nu este încă declarată ! Translatarea procedurii Q ca procedură proprie (subprogram propriu) pentru procedura P, după modelul :

```
procedure P (lista de parametri formali);
  const
    . . .
  var
    . . .
  procedure Q (lista parametri formali);
    . . .
    begin {corp procedura Q}
      . . .
      P (lista parametri actuali); {apelarea procedurii P}
      . . .
    end; {sfârşit procedura Q}
  begin {corp procedura P}
    . . .
    Q (lista de parametri actuali); {apelarea procedurii Q}
    . . .
  end; {sfârşit procedura P}
```

nu rezolvă problema întrucât acum nu va fi cunoscută procedura P la nivelul procedurii Q ce o utilizează ! Evident, rolurile procedurilor P și Q pot fi inversate, problemele rămânând aceleași.

Pentru a ieşi din acest cerc vicios ce apare în cazul declarării subprogramelor indirect recursive, limbajul PASCAL prevede un mijloc de definire ulterioară a unei proceduri sau a unei funcţii, asigurat de directiva **forward**. Folosind cuvântul **forward**, antetul unei proceduri (funcţii) poate fi scris separat de corpul acesteia (în ordinea scrierii, localizat în program dincolo, cel puţin, de declaraţia unei alte proceduri sau funcţii).

Practic, se declară mai întâi numele şi parametrii formali ai subprogramului, acest antet fiind urmat imediat de directiva **forward** iar apariţia ulterioară a corpului subprogramului este prefixată doar de numele subprogramului, fără a se mai specifica parametrii săi formali. În exemplul de mai jos este redat modul de utilizare a directivei **forward** :

```
program Exemplu ;
   const
   ...
   procedure Q (lista de parametri formali) ; forward;
   procedure P (lista de parametri formali) ;
      ...
      begin
        Q (lista de parametri actuali) ; {apelarea procedurii Q în procedura P}
        ...
      end;
   procedure Q; {nu se mai precizează parametrii formali}
      ...
      begin
        ...
      end;
   begin
      ...
   end.
```

EXERCIŢII

1. Să se scrie o funcţie recursivă şi o alta iterativă pentru calculul valorilor polinoamelor Hermite $H_n(x)$, ştiind că :

$H_0(x) = 1$

$H_1(x) = 2x$

$H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$ pentru $n > 1$

2. Pentru N citit de la tastatură, să se calculeze suma

$$S = \frac{1}{2} + \frac{1*3}{2*4} + \ldots + \frac{1*3*\ldots*(2*N-1)}{2*4*\ldots*(2*N)}$$

3. Să se calculeze coeficienţii binomiali $C_n^1, C_n^2, \ldots C_n^p$, în care n şi p sunt întregi pozitivi citiţi de la tastatură ($p <= n$), ştiind că există următoarea relaţie de recurenţă:

$$C_n^k = \frac{n-K+1}{K} C_n^{k-1}$$

$C_n^0 = 1$.

4. Problema săriturii calului. Să se realizeze un program care tipăreşte drumul parcurs de un cal pe o tablă de şah de dimensiuni m*n pentru a parcurge toate poziţiile tablei fără a trece de două ori printr-o poziţie.

5. Problema celor 8 regine. Să se scrie un program care să determine pozițiile pe tabla de șah de dimensiune 8*8 a 8 regine, astfel încât ele să nu se poată captura. (În șah, o regină capturează orice piesă care se găsește pe linia, pe coloana sau pe una din cele două diagonale ce trec prin poziția ocupată de regină).

6. Variantă a problemei celor 8 regine. Să se scrie programul care determină toate pozițiile celor 8 regine în care acestea nu se pot captura.

7. Turnurile din Hanoi. Se dau trei etaje (STÂNGA, MIJLOC, DREAPTA) și N discuri de diferite dimensiuni stivuite pe tija STÂNGA, în ordinea descrescătoare a dimensiunilor lor, formând un „turn". Se cere să se mute cele N discuri din poziția sursă (STÂNGA) în poziția destinație (DREAPTA) respectând următoarele reguli :

— în fiecare mișcare se mută un singur disc ;
— un disc nu poate fi plasat peste unul mai mic ;
— tija MIJLOC se folosește ca poziție intermediară.

CAPITOLUL 12

ALTE TIPURI STRUCTURATE DE DATE

În capitolul referitor la tipurile definite de utilizator au fost menţionate ca făcând parte din tipurile structurate de date : tabloul, înregistrarea, mulţimea, fişierul şi s-au prezentat detalii despre tablouri (array). În continuare sunt discutate celelalte trei categorii şi anume : *înregistrarea, fişierul, mulţimea.*

12.1. TIPUL ÎNREGISTRARE (RECORD)

O **înregistrare** este o *colecţie de date referitoare la o anumită entitate.* Ea poate fi folosită ca un întreg sau pot fi folosite numai anumite elemente ce o compun, denumite **câmpuri.** Fiecărui câmp i se asociază un nume cu ajutorul căruia poate fi referit. De exemplu, data calendaristică poate fi considerată ca fiind o înregistrare ce conţine trei câmpuri : ziua, luna şi anul. Ea poate fi folosită ca un întreg, drept răspuns la o întrebare de tipul „În ce dată sîntem astăzi ?" sau se pot folosi numai anumite componente ale sale, drept răspuns la o întrebare de tipul „În ce lună suntem ?"

Noţiunea de înregistrare este deosebit de utilă în cazul în care se lucrează cu înregistrări asemănătoare care conţin date despre mai multe obiecte.

De exemplu, un registru de evidenţă a populaţiei va cuprinde mai multe înregistrări cu aceeaşi structură de cîmpuri (nume, prenume, data naşterii, stare civilă) pentru mai multe persoane diferite.

Observaţie : Înregistrările nu trebuie confundate cu tablourile !

O **înregistrare** reprezintă o colecţie de componente (câmpuri) care pot fi de tipuri diferite. Fiecare câmp are propriul său nume.

Un **tablou** reprezintă o *colecţie de componente (elemente) care sunt toate de acelaşi tip.* Elementele tabloului nu poartă nume distincte ci sunt identificate cu ajutorul unui index care arată ce poziţie ocupă acestea în tablou. În cazul unui tablou ne putem referi la elementul de pe poziţia i, ceea ce nu este posibil în cazul unei înregistrări.

În limbajul PASCAL, tipul înregistrare se specifică folosind o sintaxă asemănătoare celei folosite pentru declararea variabilelor, prin scrierea între cuvintele-cheie **record** şi **end** a identificatorilor şi a tipurilor câmpurilor componente. Nu există restricţii în privinţa tipurilor câmpurilor. Referirea la un anumit câmp al înregistrării se face cu ajutorul unui **specificator de câmp.**

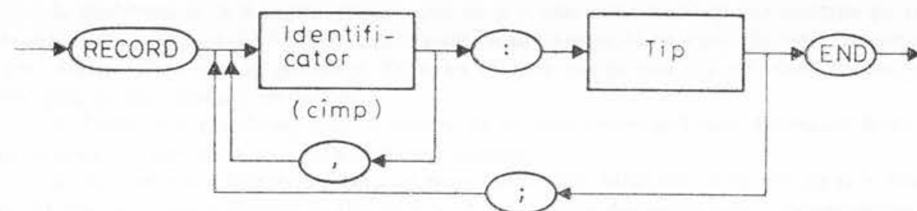


Fig. 12.1.

Diagrama de sintaxă (simplificată) a tipului **înregistrare** este prezentată în figură 12.1.

Diagrama de sintaxă a specificatorului de cîmp este prezentată în figura 12.2.

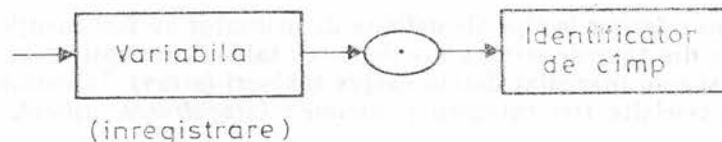


Fig. 12.2.

Exemplul 1. Definiţia tipului înregistrare

```
Date=record
     Zi:1..31;
     Luna:1..12;
     An:0..2099
   end;
```

poate fi folosită în declararea unor variabile ale căror valori reprezintă date:

Astăzi, Mâine: Date;

Fiecare dintre aceste variabile înregistrare poate fi considerată ca fiind un întreg cu trei compartimente, numite Zi, Luna, An:

Astăzi	27	1	1994
Mâine	28	1	1994

Astăzi. Luna este un specificator de câmp care desemnează câmpul Luna al variabilei înregistrare Astăzi. În exemplul prezentat, câmpurile au următoarele valori:

Astăzi.Zi 27

Astăzi.Luna 1

Astăzi.An 1994

Specificatorii de câmp pot fi folosiţi şi în scrierea instrucţiunilor. De exemplu, instrucţiunea:

Mâine.Zi : = Astăzi.Zi + 1

va reactualiza valoarea câmpului Zi al variabilei înregistrare Mâine.

Se spune că într-un specificator de câmp cum ar fi, de exemplu, Astăzi.Zi, identificatorul câmpului (Zi) este **calificat** de către variabila înregistrare (Astăzi).

În interiorul fiecărui tip înregistrare, fiecărui câmp i se asociază un identificator unic. Din punctul de vedere al domeniului de vizibilitate al identificatorului, tipul înregistrare se comportă ca un bloc : identificatorii asociați câmpurilor sunt locali în definiția de tip înregistrare și inexistenți în exterior. Accesul la acest domeniu, aparent închis, se realizează prin procedeul descris anterior și numit **calificare.** De aceea, în program se pot folosi identificatori identici pentru câmpuri și : constante, tipuri, variabile, proceduri sau chiar câmpuri ale altor tipuri înregistrare. Datorită cerinței de calificare a câmpurilor este prevenită orice posibilă ambiguitate.

Exemplul 2.

```
var A:array [2..8] of integer;
    A:2..8;
```

este o *formulare nepermisă* : definirea variabilei A este ambiguă.

```
var A:integer;
    B:record A:real;
             B:boolean
      end;
```

este o *formulare corectă* deoarece variabila A de tip integer nu poate fi confundată cu B.A de tip real iar variabila înregistrare B este distinctă de B.B de tip logic.

Până în acest moment s-a arătat că se pot folosi câmpurile individuale ale unei înregistrări în același mod în care sunt utilizate orice alte variabile. Un avantaj important al structurării datelor ca înregistrare îl constituie folosirea acestora ca un întreg fără a mai fi preocupați de câmpurile componente.

Între înregistrările de același tip, limbajul PASCAL permite operații de atribuire și aplicarea operatorilor relaționali = și <>.

Exemplul 3.

Considerând tipul Date definit în exemplul 1 și declarând variabilele :

```
Astăzi, DataScrisoare: Date;
```

atunci instrucțiunea următoare:

```
DataScrisoare:=Astăzi;
```

va determina copierea întregii înregistări Astăzi în variabila DataScrisoare, operație echivalentă cu următoarele :

```
Data Scrisoare.Zi  :=Astăzi.Zi;
DataScrisoare.Luna :=Astăzi. Luna;
DataScrisoare.An   :=Astăzi.An;
```

Observație : După cum s-a precizat deja, în cazul operațiilor efectuate asupra înregistărilor complete trebuie respectată **regula compatibilității din punctul de vedere al tipului.**

Exemplul 4. Declarând variabilele :

```
NrComplex: record
             Re, Im:real;
           end;

X1, X2    :record
             Re, Im:real;
           end;
```

se consideră că variabilele X1 şi X2 sunt de acelaşi tip, deoarece ele au fost declarate împreună. Fiind declarată separat, variabila NrComplex va fi considerată a fi de tip diferit, astfel încât următoarele instrucţiuni:

```
X1:=X2;
X2:=X1;
```

sunt corecte, în timp ce instrucţiunile:

```
NrComplex:=X1;
X1:=NrComplex;
X2:=NrComplex;
NrComplex:=X2
```

sunt incorecte.

Înregistrările complete pot fi folosite şi ca parametri valoare şi/sau parametri variabilă ai subprogramelor.

Exemplul 5.

Structura înregistrărilor conţinute în registrul de stare civilă poate fi descrisă, într-o formă simplificată, cu următoarea definiţie de tip:

```
DatePersonale=record
                Nume, Pronume:packed array [1...12] of char;
                DataNastere :Date;
                Sex         :(M, F);
                StareCivilă :(NC, C, D, V)
            end;
```

Se datorește scrierea unei proceduri care, inspectând registrul de stare civilă precizează dacă un anumit bărbat îndeplineşte condiţiile legale pentru a se căsători cu o anumită femeie, şi, dacă acest lucru este posibil, actualizează în mod corespunzător înregistrările.

```
procedure Căsătorii (var Soţ, Soţie:DatePersonale;
                     var Legal:boolean);
  begin
    Legal:=(Soţ.Sex=M) and (Soţie.Sex=F) and
           (Soţ.StareCivilă<>C) and (Soţie.StareCivilă<>C);
    if Legal then {reactualizare înregistrări}
      begin
        Soţ.StareCivilă:=C;
        Soţie.StareCivilă:=C;
      end
  end. {Căsătorii}
```

Parametrii actuali ce corespund cu Soţ şi Soţie trebuie să fie variabile înregistrare de acelaşi tip ca şi parametrii formali. De exemplu, dispunând în programul apelant de variabilele:

```
Constantin, Veronica : DatePersonale;
Permis               : boolean
```

se putea scrie instrucţiunea:

```
Căsătorii (Constantin, Veronica, Permis);
```

12.1.1. Instrucţiunea WITH

Uneori este necesar ca în cadrul programului să fie scrise mai multe instrucţiuni ce se referă la câmpurile unei aceeaşi variabile înregistrare, ceea ce conduce la repetarea numelui variabilei de fiecare dată când se realizează o calificare a câmpului. Limbajul PASCAL permite scrierea într-o formă prescurtată a acestor referiri cu ajutorul instrucţiunii **with**.

Exemplul 6. Folosind tipul declarat în exemplul 5, se declară variabila:
DateDv:DatePersonale;
Următorul fragment de program:

```
write(DateDv.Prenume,' ', DateDv.Nume,'este');
case DateDv.Sex of
      M:write('bărbat');
      F:write('femeie')
end;
```

poate fi scris pe scurt astfel:

```
with DateDv do
    begin write (Prenume,' ', Nume, 'este');
          case Sex of
           M:write('bărbat');
           F:wiite('femeie')
          end {case}
    end; {with}
```

Prin scrierea variabilei înregistrare (DateDv) între cuvintele-cheie **with** și **do**, nu mai este necesar ca în interiorul instrucțiunii **with** să calificăm identificatorii câmpurilor înregistrării.

Diagrama de sintaxă simplificată a instrucțiunii **with** este prezentată în figura 12.3.

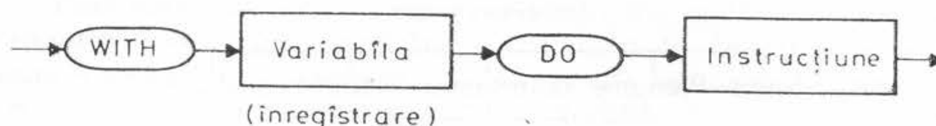


Fig. 12.3.

În interiorul unei instrucțiuni **with**, orice identificator necalificat care este identificator de câmp al variabilei înregistrare reprezintă în mod real acel câmp. Singurul loc în care poate fi folosit un identificator de câmp, fără a fi calificat, este interiorul unei instrucțiuni **with**.

Instrucțiunea ce formează „corpul" instrucțiunii **with** se comportă ca un bloc în cadrul căruia se poate considera că identificatorii de câmp ai înregistrării au fost declarați la fel ca și variabilele obișnuite. Pentru stabilirea domeniului de valabilitate al identificatorilor, se aplică deci regulile enunțate în cadrul capitolului 10.

Atunci când se dorește prelucrarea mai multor înregistări, pot fi prescurtați folosind instrucțiunea **with** numai specificatorii câmpurilor ce aparțin unei singure înregistrări, așa cum rezultă din exemplul următor.

Exemplul 7. Funcție pentru calcularea vârstei în ani a unei persoane, cunoscând datele personale ale acesteia și data curentă, memorată în variabila globală Astăzi.
Scrierea completă este:

```
function Vârsta (Persoana:DatePersonale):integer;
  begin
     if (Astăzi.Luna > Persoana.DataNaştere.Luna) or
        (Astăzi.Luna=Persoana.DataNaştere.Luna) and
        (Astăzi.Zi >=Persoana.DataNaştere.Zi)
     then Vârsta:=Astăzi.An — Persoana.DataNaştere.An
     else Vârsta:=Astăzi.An — Persoana.DataNaştere.An — 1
```

```
end; {Vârsta} ;
```

în timp ce scrierea prescurtată este :

```
function Vârsta (Persoana:DatePersonale):integer;
 begin
   with Persoana.DataNaştere do
     if (Astăzi.Luna > Luna) or
        (Astăzi.Luna=Luna) and
        (Astăzi.Zi >=Zi)
     then Vârsta:=Astăzi.An — An
     else Vârsta:=Astăzi.An — An — 1
 end; {Vârsta};
```

Acolo unde nu sunt calificate, câmpurile Zi, Luna, An se referă la câmpurile ce fac parte din Persoana.DataNaştere.

Observaţie : O înregistrare ale cărei câmpuri sunt ele însele înregistrări este numită uneori **înregistrare ierarhizată**. În exemplul anterior s-a folosit o astfel de înregistrare, DatePersonale, al cărui câmp DataNaştere este el însuşi o înregistrare, de tipul Date.

Structura ierarhizată a tipului DatePersonale este ilustrată în figura 12.4. De multe ori, înregistările ierarhizate se prelucrează cu ajutorul unor instrucţiuni **with** imbricate, ca în exemplul următor :

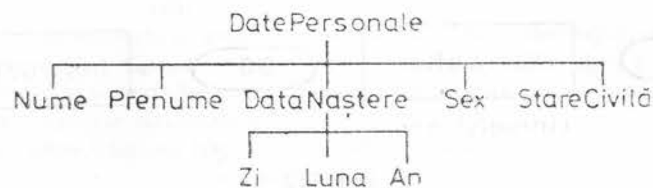


Fig. 12.4.

Exemplul 8.

```
with DateDv do
  begin
    Sex:=M;
    StareCivilă:=NC;
    with DataNaştere do
      begin
        Zi:=30;
        Luna:=6;
        An:=1963
      end
  end;
```

Înregistările pot fi folosite şi pentru scrierea tabelelor, o structură de date des folosită în practică şi care conţine, în general, mai multe date asociate fiecărei intrări (fiecare intrare poate constitui o înregistrare).

Exemplul 9.

Tabela numelor mai multor oraşe şi a numărului de locuitori ai acestora poate fi definită astfel :

```
oraş:array [1..MaxOraşe] of
```

```
record
    Nume:packed array [1..20] of char;
    Locuitor:integer
end;
```

Numărul de locuitori ai orașului cu numărul N va fi desemnat de:
oraș[N].Locuitori

Și în cazul tipului înregistrare se poate folosi metoda de împachetare a datelor, aceasta fiind precizată cu ajutorul cuvântului-cheie **packed** ce prefixează definiția. De exemplu, pentru a economisi spațiu de memorie, tipul Date poate fi redefinit astfel:

```
Date=packed record
    Zi:1..31;
    Luna:1..12;
    An:0..2099
end
```

Toate celelalte considerații referitoare la această metodă, expuse în cadrul capitolului 9, sunt valabile și în acest caz.

12.1.2. Înregistrarea cu variante

Până acum am discutat despre înregistrări fixe ca structură internă, înregistrările aceluiași tip conținând același număr de câmpuri, cu aceleași nume și aceleași tipuri. Limbajul PASCAL permite însă și folosirea unor înregistrări cu variante, cu o structură mai flexibilă.

Exemplul 10. Pentru fiecare carte sau disc, informațiile cuprinse în catalogul unei biblioteci se referă la:

— numărul de înregistrare;
— titlu;
— cod autor/compozitor;
— nume editură;
— tip de articol: carte sau disc;
— număr ediție (numai pentru cărți);
— anul publicării (numai pentru cărți);
— nume interpret (numai pentru discuri).

Informațiile pot fi reprezentate cu ajutorul unei înregistări formate din opt câmpuri dar în fiecare caz se folosesc în mod real doar șase sau șapte câmpuri. Prin folosirea unei înregistrări cu variante, câmpurile ce vor fi conținute în mod real în fiecare înregistrare sunt precizate de către câmpul corespunzător tipului de articol (carte sau disc):

```
TipArticol=(Carte, Disc);
Articole=record
        NI:0..30000;
        Titlu:packed array [1..30] of char;
        Autor:packed array [1..16] of char;
        Editura:packed array [1..20] of char;
        case Tip:TipArticol of
            Carte: (Ediție:1..50; An:0..1999);
            Disc: (Interpret:packed array [1..20] of char)
        end;
```

Prin scrierea condiției „**case** Tip : TipArticol **of**" se definește un câmp special, numit **câmp selector,** de tip TipArticol, ceea ce conferă fiecărei înregistrări de tip TipArticol o proprietate aparte: numărul și tipurile câmpurilor care sunt specificate **după** câmpul selector depind de valoarea curentă a acestuia. Deci, pentru valoarea Carte, cîmpul selector va fi urmat de două câmpuri, Ediție și An, iar pentru valoarea Disc el este urmat de un singur cîmp, Interpret.

Tipul câmpului selector trebuie specificat ca identificator de tip și poate fi oricare tip scalar, mai puțin tipul real.

● **Reguli ce trebuie respectate în cazul folosirii unei înregistrări cu variante :**

— pentru fiecare valoare a tipului câmpului selector trebuie specificată o variantă ; pentru valorile care nu au asociate câmpuri suplimentare se folosesc variante vide ;

— câmpul selector poate fi primul și unicul câmp al unei înregistrări ;

— câmpul selector poate fi ultimul câmp al unei înregistrări ;

— o aceeași variantă poate fi pusă în corespondență cu mai multe valori ale câmpului selector.

Exemplul 11. Se dorește crearea evidenței persoanelor înscrise pentru ocuparea unui post, precizându-se pentru fiecare persoană, numele, prenumele, data nașterii, sexul și studiile efectuate. În acest scop se pot defini următoarele tipuri:

```
type ŞirCaractere=packed array [1..25] of char;
     FormaInv=(Fără, Prof, Lic, Sup);
     DatePersonale=record
                     Nume,Prenume:ŞirCaractere;
                     DataNaştere:Date;
                     Sex: (M, F);
                     case Studii:FormaInv of
                       Fără:();
                       Prof, Lic: (Medie:real);
                       Sup: (Inst, Fac:ŞirCaractere; Medie:real;
                             Titlu: (Nimic, Dr));
                   end;
```

În exemplul prezentat, tipul FormaInv reprezintă studiile efectuate de persoana respectivă și cuprinde patru valori : „Fără", cu semnificația de „fără liceu", „Prof" cu semnificația de „învățământ profesional", „Lic" cu semnificația de „învățământ liceal" și „Sup" cu semnificația de „învățământ superior". S-a presupus că pentru un absolvent de liceu sau școală profesională se cere specificarea mediei de absolvire iar pentru un absolvent de învățământ superior se cere specificarea numelui institutului absolvit, a mediei generale și, eventual, a titlului științific obținut ulterior.

Diagrama de sintaxă a tipului înregistrare este prezentată în figura 12.5.

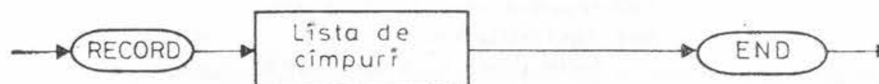


Fig. 12.5.

Diagrama de sintaxă pentru Lista câmpuri este prezentată în figura 12.6.

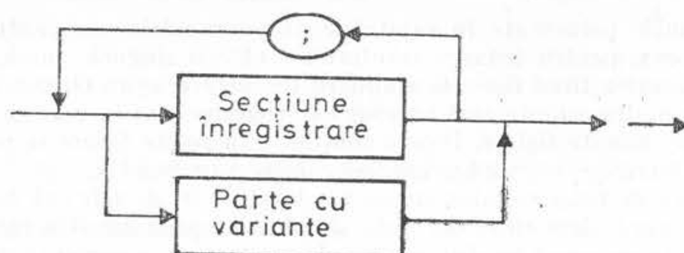


Fig. 12.6.

Diagrama de sintaxă pentru Secţiunea înregistrare este prezentată în figura 12.7.

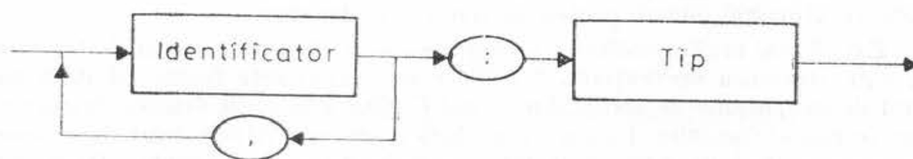


Fig. 12.7.

Diagrama de sintaxă pentru Secţiunea selectare este prezentată în figura 12.8.

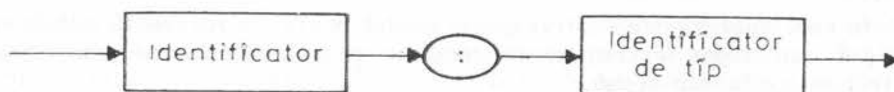


Fig. 12.8.

Diagrama de sintaxă pentru Parte cu variante este prezentată în figura 12.9.

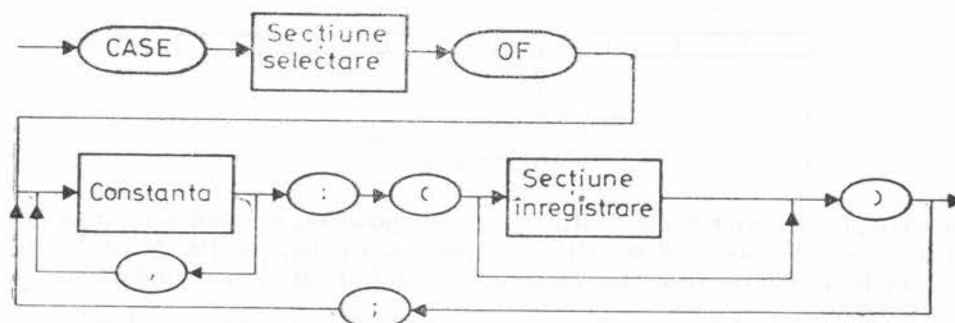


Fig. 12.9.

Programele prezentate în capitolele anterioare foloseau pentru introducerea datelor şi pentru listarea rezultatelor câte o singură sursă, respectiv destinaţie, acestea fiind fişierele standard de intrare/ieşire (Input şi Output). De cele mai multe ori este însă necesar ca programele să lucreze cu mai multe *seturi de date*, numite **fişiere**. Datele conţinute în aceste fişiere se pot memora în formă binară pe un mediu magnetic (disc sau bandă).

Noţiunea de fişier se foloseşte cu înţelesul de *set de date sub formă de text sau formă binară, date citite sau scrise de către un program*. Un fişier poate fi un set de date memorat pe disc sau bandă magnetică, o secvenţă de caractere introduse de la tastatură, conţinutul ecranului unui dispozitiv de afişare sau rezultate tipărite pe hârtie.

Există, desigur, şi restricţii în modul de operare al programului asupra unui anumit fişier. Un fişier memorat pe suport magnetic poate fi scris şi poate fi citit. Pe de altă parte, conţinutul ecranului unui dispozitiv de afişare poate fi folosit numai pentru evidenţierea rezultatelor în timp ce tastatura poate fi utilizată numai pentru introducerea datelor.

Există mai multe posibile structuri de fişiere. Cea mai simplă dintre acestea este **structura secvenţială**. Un fişier secvenţial este format dintr-o *secvenţă de componente de acelaşi tip ce pot fi citite numai în ordinea în care ele apar în cadrul fişierului*. De exemplu, dacă se doreşte citirea celei de-a zecea componente, trebuie citite mai întâi primele nouă componente. Dacă programul doreşte să revină la prima componentă, el trebuie să reînceapă citirea de la începutul fişierului. Tot succesiv se face şi scrierea unui fişier secvenţial.

Există şi structuri de fişiere care permit programului **accesul direct** la anumite componente ale fişierului, dar acestea nu fac obiectul prezentului capitol.

În continuarea, este ilustrat grafic modul în care se realizează citirea sau scrierea unui fişier secvenţial, indicându-se poziţia următoarei componente ce trebuie citită sau scrisă.

12.2.1. Citirea unui fişier secvenţial

Un exemplu de citire a unui fişier secvenţial este prezentat în figura de mai jos.

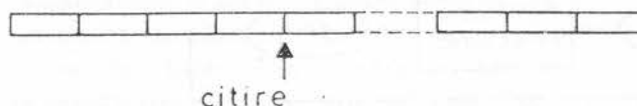


Fig. 12.10, a

În exemplul anterior au fost citite patru componente, urmând a fi citită cea de-a cincea. Atunci când se iniţiază citirea unui fişier, poziţia de citire este situată la începutul fişierului (deoarece nu a fost citită deocamdată nici o componentă).

După citirea ultimei componente a fişierului, poziţia de citire se află la sfârşitul fişierului (End-Of-File), aşa cum se arată în figura 12.10, b.

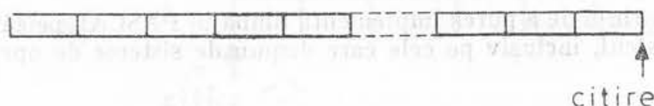


Fig. 12.10, *b*.

Dacă programul încearcă să mai citească o componentă, el va eșua.

12.2.2. Scrierea unui fișier secvențial

Poziția de scriere va fi întotdeauna la sfârșitul fișierului, acesta fiind singurul loc în care se pot scrie componentele (fig. 12.11, a).

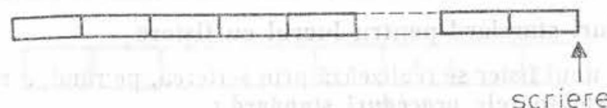


Fig. 12.11, *a*.

Scrierea unei noi componente mărește dimensiunea fișierului:

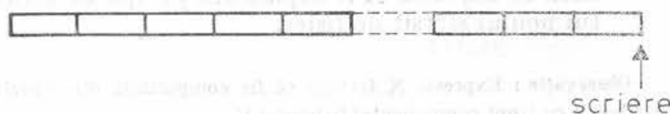


Fig. 12.11, *b*.

Fișierele pot fi comparate cu alte structuri de date, cum ar fi, de exemplu, tablourile. Ambele structuri au toate componentele de același tip. Însă, în timp ce tablourile au dimensiune fixă, fișierele secvențiale își pot extinde dimensiunea prin scrierea de noi componente. Elementele tablourilor pot fi accesate în mod aleator, în timp ce componentele unui fișier secvențial pot fi accesate numai câte una, în ordinea serială.

În limbajul PASCAL, fișierul se aseamănă cu o variabilă obișnuită : i se asociază un identificator și este declarat într-o declarație de variabilă. Tipul său precizează faptul că este vorba despre un fișier și specifică tipul componentelor acestuia. Diagrama de sintaxă a tipului fișier este prezentată în figura 12.12.

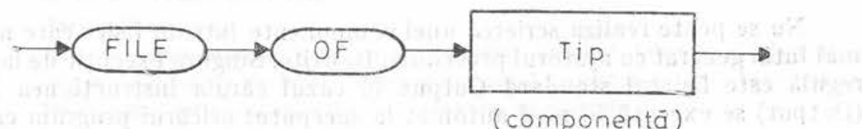


Fig. 12.12.

Tipul componentei poate fi *oricare tip admis de limbajul PASCAL, cu excepția tipului fișier sau a unui tip ce conține un tip fișier.* Cu alte cuvinte, fișierele nu pot avea drept componente alte fișiere, această restricție fiind

impusa din dorința de a putea implementa limbajul PASCAL pe cât mai multe sisteme de calcul, inclusiv pe cele care dispun de sisteme de operare extrem de simple.

Exemplul 12. Pentru păstrarea combinațiilor cromatice care apar în drapelele unor state, culorile pot fi reprezentate cu ajutorul tipului enumerare:

Culori=(Alb, Albastru, Verde, Galben, Roşu, Portocaliu, Negru);

Fiecare combinație de culori este memorată într-un fişier, iar programul selectează drapelul unui stat prin citirea fişierului asociat.

Tipul fişierului este:

Drapel=**file of** Culori;

iar drapelele statelor sunt declarate ca variabile fişier:

România, Franța, Italia, Brazilia, Germania: Drapel;

12.2.3. Proceduri standard pentru lucrul cu fişiere

Generarea unui fişier se realizează prin scrierea, pe rând, a componentelor sale folosind următoarele **proceduri standard**:

Rewrite (F) — pregăteşte pentru (re)scriere fişierul F, înlocuindu-l cu un fişier vid (care nu conține nici o componentă) și stabilind poziția inițială de scriere.

Write (F, X) — adaugă în fişierul F o componentă a cărei valoare este dată de expresia X şi deplasează poziția de scriere în dreptul noului sfârşit de fişier.

Observație: Expresia X trebuie să fie compatibilă din punctul de vedere al atribuirii cu tipul componentei fişierului F.

Efectul apelării acestei proceduri este ilustrat în figura 12.13.

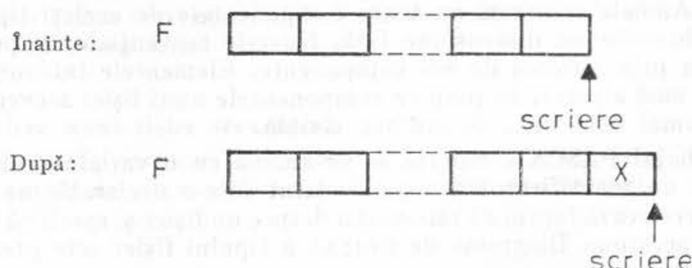


Fig. 12.13.

Nu se poate realiza scrierea unei componente într-un fişier care nu a fost mai întâi generat cu ajutorul procedurii Rewrite. Singura excepție de la această regulă este fişierul standard Output în cazul căruia instrucțiunea Rewrite (Output) se execută în mod automat la începutul oricărui program care conține ca parametru Output.

Un fişier existent poate fi citit de către programul care l-a scris sau de către alte programe, folosind următoarele **proceduri standard**:

Reset (F) — pregăteşte fişierul F pentru citire, mutând poziția de citire la începutul fişierului. Efectul apelării acestei proceduri este ilustrat în figura 12.14.

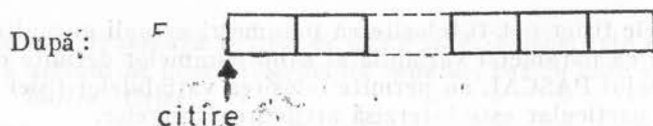


Fig. 12.14.

Read (F, V) — copiază următoarea componentă a fişierului F în variabila V şi apoi avansează poziţia de citire dincolo de această componentă.

Observaţie: Componentele fişierului F trebuie să fie compatibile din punctul de vedere al atribuirii cu tipul variabilei V.

Efectul apelării acestei proceduri este prezentat în figura 12.15.

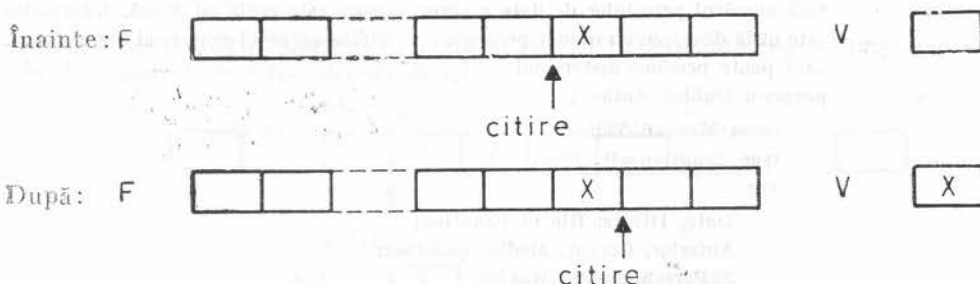


Fig. 12.15.

Nu se poate realiza citirea unui fişier, dacă acesta nu a fost pregătit mai întâi cu ajutorul procedurii Reset. Singura excepţie de la această regulă este fişierul standard Input în cazul căruia instrucţiunea Reset(Input) se execută în mod automat la începutul oricărui program care conţine ca parametru Input.

În cazul în care fişierul F se găseşte deja la sfârşit de fişier, instrucţiunea Read(F, V) va eşua. Pentru a preîntâmpina o astfel de situaţie se foloseşte funcţia standard logică EOF(F). În cazul ultimei diagrame prezentate ca exemplu, EOF(F) are valoarea False dar ea va primi valoarea True după citirea a încă două componente.

Observaţii:

1. Dacă se doreşte lucrul cu fişiere stocate pe suport magnetic extern (disc magnetic) se va folosi procedura:

Assign (F, Nume);

în care Nume reprezintă un şir de caractere încadrat între apostrofuri şi F este o variabilă de tip fişier. Astfel se realizează asocierea numelui fişierului extern (Nume) variabilei fişier (F). Se poate scrie, de exemplu:

```
Assign (F,' note.dat');
Assign (G, 'c:/lucru/rez.dat');
```

2. Închiderea unui fişier se efectuează în urma execuţiei instrucţiunilor de tip Close(F). La terminarea programului, în mod automat are loc închiderea tuturor fişierelor utilizate, chiar dacă acest lucru nu a fost cerut în mod explicit cu ajutorul procedurii Close.

Variabilele fişier pot fi folosite ca parametri actuali ai multor proceduri standard sau ca parametri variabilă ai subprogramelor definite de către utilizator. Limbajul PASCAL nu permite folosirea variabilelor fişier complete în alt mod ; în particular este interzisă atribuirea fişierelor.

Toate variabilele fişier (parametri sau nu) trebuie declarate în modul obişnuit. Excepţie de la această regulă fac din nou fişierele standard Input şi Output care nu trebuie declarate, chiar dacă sunt folosite.

Exemplul 13. O piesă muzicală poate fi înregistrată digital, măsurând cu ajutorul unui microfon intensitatea semnalului şi reprezentând-o apoi ca număr cuprins în domeniul 0—65535. Eşantionând sunetul în fiecare secundă se poate memora şi reproduce sunetul Hi—Fi. Inginerii de sunet folosesc de două ori mai multe măsurători pe secundă. Următorul program va considera ca date de intrare fişierul semnalelor audio digitalizate şi va dubla numărul de măsurători prin plasarea între fiecare două valori a mediei acestora. De asemenea, va contoriza numărul perechilor de date a căror valoare este egală cu 65535. Informaţia este utilă deoarece un număr prea mare va indica un nivel excesiv al înregistrării, care poate produce distorsiuni.

```
program DubluEsantion ;
    const Max=65535;
    type Esantion=0..Max;
    var
        Date, DDate: file of Esantion;
        Anterior, Curent, Medie: Esantion;
        NrPerechiMax:0..MaxInt;
    begin
        assign (Date, 'Masuri.dat');
        assign (DDate, 'Rezult.dat');
        reset (Date);
        rewrite (DDate);
        NrPerechiMax:=0;
        read (Date, Anterior);
        write (DDate, Anterior);
        while not EOF (Date) do
          begin
            read (Date, Curent);
            Medie:=(Anterior +Curent) div 2;
            if Medie=Max then
              NrPerechiMax:=NrPerechiMax+1;
            write (DDate, Medie);
            write (DDate, Curent);
            Anterior:=Curent
          end;
        writeln ('Nr. perechilor de intensitate maximă a fost:',
                          NrPerechiMax:2)
    end.
```

12.2.4. Buffere fişier

Bufferele fişier apar numai în variantele standard ale limbajului.

În cadrul prezentării procedurii standard Read(F, V), s-a arătat că ea are drept efect două acţiuni : copiază următoarea componentă a lui F în va-

riabila V și apoi avansează poziția de citire în F. Uneori este necesar însă ca aceste două acțiuni să poată fi generate separat, caz în care se folosește noțiunea de **buffer fișier.**

Declararea oricărei variabile fișier F va introduce în mod automat o **variabilă buffer**, notată F↑, de același tip cu tipul componentelor, considerată ca fiind o *„fereastră"* prin care se pot inspecta (citi) componentele deja existente sau se pot adăuga (scrie) noi componente și care poate fi deplasată cu ajutorul anumitor operatori.

Observație : Prelucrarea secvențială a informațiilor și existența unei variabile buffer sugerează asocierea fișierelor cu memoria secundară și dispozitivele periferice. Modul exact de realizare a alocării componentelor depinde de implementare dar se presupune că în memoria primară, la un moment dat, există numai anumite componente și că este direct accesibilă doar componenta indicată de către F↑.

În cazul citirii fișierului F, bufferul fișier F↑ reprezintă copia componentei lui F situate în poziția curentă de citire :

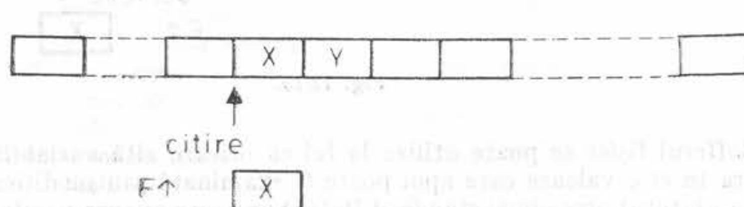


Fig. 12.16.

Bufferul fișier poate fi folosit în același mod ca și oricare alta variabilă ; în particular, se poate examina în mod direct valoarea pe care o conține, fără a fi necesar ca aceasta să fie copiată în altă variabilă.

Cu ajutorul procedurii standard Get(F) se poate avansa poziția de citire, reactualizând în mod automat bufferul fișier. Efectul apelării acestei proceduri asupra fișierului F ilustrat în figura anterioară este redat în figura 12.17.

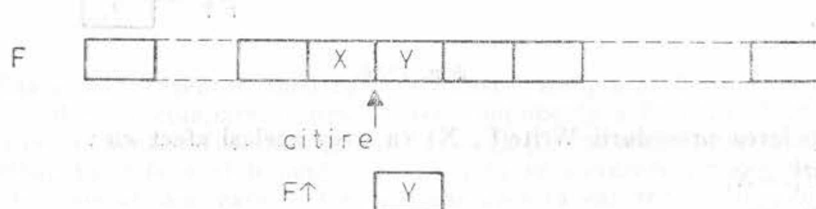


Fig. 12.17.

Apelarea procedurii Read(F, V) va avea același efect cu :

```
begin V:=F↑;
   Get(F)
end ;
```

Prin apelarea procedurii Reset(F) se realizează copierea primei componente a fișierului F în bufferul fișier F↑ : așa cum se reprezintă în figura 12.18.

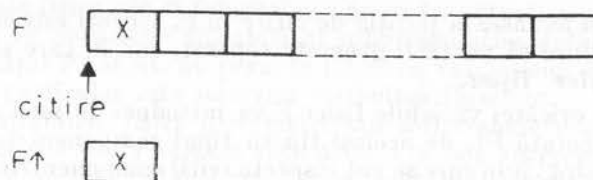


Fig. 12.18.

Bufferul fişier apare şi în cazul în care se doreşte scrierea într-un fişier. Fişierului i se asociază un astfel de buffer în care se va memora valoarea fiecărei noi componente până în momentul avansării poziţiei de scriere (fig. 12.19).

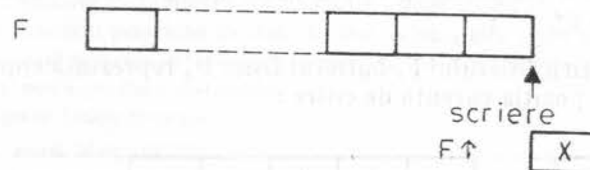


Fig. 12.19.

Bufferul fişier se poate utiliza la fel ca oricare altă variabilă : se poate memora în el o valoare care apoi poate fi examinată sau modificată.

Cu ajutorul procedurii standard Put(F) se poate avansa poziţia de scriere, adăugând conţinutul bufferului fişier în fişier şi lăsând nedefinit acest buffer (fig. 12.20).

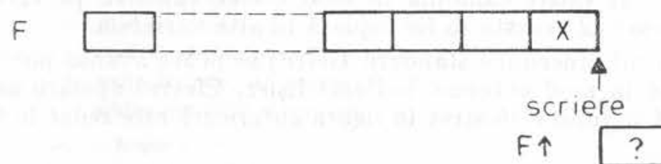


Fig. 12.20.

Apelarea procedurii Write(F, X) va avea acelaşi efect cu :

```
begin F↑:=X;
    Put(F)
end;
```

Exemplul 14. Următorul fragment de program operează asupra a două fişiere care conţin secvenţe ordonate de numere întregi : $f_1, f_2, \ldots, f_m$ şi $g_1, g_2, \ldots, g_n$, în care $f_{i+1} >= f_i$ şi $g_{j+1} >= g_j$, pentru toţi i, j şi combină aceste două fişiere într-un singur fişier ordonat, h, astfel încât $h_{k+1} >= h_k$, pentru $k = 1, 2, \ldots, (m + n - 1)$. Programul va utiliza variabilele :

```
endfg:boolean;
f, g, h: file of integer;
```

și va avea următoarea structură:

```
begin
  reset (f);
  reset (g);
  rewrite (h);
  endfg:=eof(f) or eof(g) ;
  while not endfg do
    begin
      if f↑ < g↑ then
        begin
          h↑:=f↑;
          get (f);
          endfg:=eof (f)
        end
      else
        begin
          h↑:=g↑;
          get (g);
          endfg:=eof (g)
        end;
      put (h)
    end;
  while not eof (g) do
    begin
      h↑:=g↑;
      put (h);
      get (g)
    end;
  while not eof (f) do
    begin
      h↑:=f↑;
      put (h);
      get (f)
    end;
end.
```

Exemplul prezentat ilustrează utilitatea bufferului fișier. În fiecare etapă se dorește compararea următoarei componente a fișierului f cu următoarea componentă a fișierului g, cea mai mică (în valoare) dintre ele urmând a fi adăugată în fișierul h pentru ca apoi să se avanseze poziția de citire doar în fișierul din care a fost extrasă această valoare. Componentele ce trebuie comparate se află în bufferele fișier f↑ și g↑ care pot fi examinate fără a avansa poziția de citire în fișierele corespunzătoare.

12.2.5. Fișiere text

Fișierele text sunt fișierele ale căror *componente sunt caractere și marcatori de sfârșit de linie.* Acestea sunt tratate separat deoarece constituie modalitatea normală de comunicare între utilizatorul uman și sistemul de calcul.

Limbajul PASCAL permite declararea fișierelor text cu ajutorul tipului fișier standard Text. Fișierele standard Input și Output sunt fișiere de tip Text.

În mod obișnuit, textele sunt structurate în **linii.** O metodă simplă de indicare a delimitării dintre două linii consecutive este folosirea **caracterelor de control.** De exemplu, în setul de caractere ASCII se folosesc pentru marcarea sfârșitului de linie două caractere: CR (carriage return) și LF (line feed). Cu toate acestea, în multe sisteme de calcul se utilizează seturi de caractere ce nu conțin astfel de caractere de control, ceea ce face necesară folosirea altor metode pentru indicarea sfârșitului de linie.

Marcatorul de sfârșit de linie ce face parte din tipul fișier Text nu poate fi atribuit unei variabile de tip char, dar poate fi recunoscut și poate fi generat cu ajutorul următoarelor **proceduri :**

Writeln(F) — încheie linia curentă a fișierului text F;

Readln(F) — trece la începutul următoarei linii a fișierului text F (F↑ primește valoarea primului caracter al acestei linii);

EOLn(F) — funcție de tip boolean care arată dacă s-a ajuns la sfârșitul liniei curente a fișierului text F (dacă ea are valoarea True, F↑ corespunde poziției separatorului de linie, dar are valoarea spațiului).

În cazul în care F este un fișier text, iar Caracter o variabilă de tip char, pot fi folosite următoarele **notații prescurtate :**

Prescurtare	Forma extinsă
write(F, caracter)	F↑:=Caracter; put(F);
read(F, caracter)	Caracter:=F↑; get(F);

Exemplul 15. Următorul program realizează copierea unui fișier text, fără a folosi fișierele standard Input, Output:

```
program Copiere;
  var original, copie: Text;
      caracter:char;
  begin
    ...
      reset (original);
      rewrite (copie);
      while not eof (original) do
        begin
          while not eoln(original) do
            begin
              read (original, caracter);
              write (copie, caracter)
            end;
          writeln (copie);
          readln (original)
        end;
      close (copie)
  end.
```

12.3. TIPUL MULŢIME (SET)

În matematică, **mulţimea (set)** reprezintă o *noţiune de bază*. Cea mai convenabilă definiţie a unei mulţimi, din punctul de vedere al limbajului PASCAL, este : mulţimea reprezintă o *colecţie de obiecte distincte*, toate având *acelaşi tip* şi care se numesc **membri** *ai mulţimii*.

{Verde, Roşu, Galben} şi {3, 5, 10, 4} sunt două mulţimi care au fiecare trei şi, respectiv, patru membri.

Ordinea listării membrilor nu are nici o importanţă : {3, 5, 10, 4} este aceeaşi mulţime ca şi {5, 10, 3, 4}.

De asemenea, aceiaşi membri pot să apară în cadrul mulţimii de mai multe ori : {3, 3, 5, 5, 6} este aceeaşi mulţime cu {3, 5, 6}.

Se numeşte **mulţime vidă** o *mulţime fără membri*.

● **Operaţiile ce se pot realiza folosind ca operanzi mulţimi** sînt :

1. **Verificarea apartenenţei** : se stabileşte faptul că o anumită valoare este sau nu membru al unei mulţimi precizate. De exemplu, Roşu este membru al mulţimii {Verde, Roşu, Galben} în timp ce Negru nu îndeplineşte această condiţie ;

2. **Verificarea incluziunii** : o mulţime A este inclusă într-o mulţime B (sau A este o submulţime a lui B) dacă şi numai dacă fiecare membru al lui A este şi membru al lui B. De exemplu, {Verde, Roşu, Galben}, {Galben} şi mulţime vidă sunt submulţimi ale mulţimii {Verde, Roşu, Galben} ;

3. **Intersecţia mulţimilor** : intersecţia a două mulţimi A şi B, este mulţimea tuturor valorilor care sunt membri ai lui A şi, în acelaşi timp, sunt membri ai lui B. De exemplu, intersecţia mulţimilor {Verde, Roşu, Galben} şi {Alb, Verde, Portocaliu} este mulţimea {Verde} ;

4. **Reuniunea mulţimilor** : reuniunea a două mulţimi A şi B este mulţimea tuturor valorilor care sunt membri ai lui A sau ai lui B sau ai amândurora. De exemplu, reuniunea mulţimilor {Verde, Roşu, Galben} şi {Verde, Roşu, Portocaliu} este mulţimea {Verde, Roşu, Galben, Portocaliu} ;

5. **Diferenţa mulţimilor** : diferenţa a două mulţimi A şi B este mulţimea tuturor valorilor care sunt membri ai mulţimii A dar nu sunt membri ai mulţimii B. De exemplu, diferenţa mulţimilor {Verde, Roşu, Galben} şi {Alb, Albastru, Galben, Portocaliu} este mulţimea {Verde, Roşu}.

Spre deosebire de alte limbaje de programare, limbajul PASCAL permite exploatarea deplină a avantajelor eleganţei furnizate de notaţia de tip mulţime. În programele scrise se pot declara variabile mulţime şi se pot compune expresii de tip mulţime folosind operatorii descrişi anterior.

Diagrama de sintaxă a tipului mulţime este prezentată în figura 12.21.

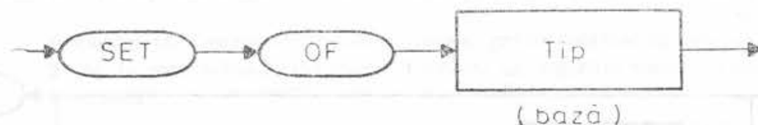


Fig. 12.21.

Set şi **of** sunt cuvinte rezervate.

Pentru a permite o implementare eficientă, limbajul PASCAL impune cerinţa ca **tipul de bază** (tipul membrilor mulţimii) să fie orice tip scalar, cu excepţia tipului real. Mai mult, dacă tipul de bază este integer (sau un subdomeniu al acestuia), compilatorul PASCAL are posibilitatea de a impune limita inferioară şi limita superioară a valorilor ce pot fi utilizate.

Observaţie : Atunci când programul se rulează pe un sistem de calcul CORAL, tipul mulţime este extins la 128 elemente, deci într-o mulţime pot fi precizaţi cel mult 128 membri. În cazul calculatoarelor compatibile IMB-PC, numărul maxim de membri ai unei mulţimi este de 256 (a se vedea versiunea Turbo-PASCAL 6.0).

De exemplu, fiind dată definirea de tip :
type CuloriBază=(Roşu,Verde,Albastru) ;
se poate declara un nou tip :
Culoare=**set of** CuloriBază ;
sau variabile mulţime :
var Culoare1,Culoare2:**set of** CuloriBază ;
De asemenea, folosind tipul mulţime Culoare se pot declara alte variabile mulţime :
var Nuanţa1, Nuanţa2:Culoare ;

Referirea unei valori de tip mulţime se realizează folosind o notaţie asemănătoare cu notaţia matematică în care se înlocuiesc acoladele cu paranteze pătrate şi se separă elementele mulţimii cu ajutorul virgulelor. În acest mod se realizează aşa-numiţii **constructori de mulţime (set constructors). De exemplu:**
Litere:=['A','B','C']
atribuie variabilei mulţime Litere o valoare mulţime care conţine trei membri. Se poate folosi şi o prescurtare a constructorului de mulţime prezentat anterior : Litere:=['A'..'C'];

Această instrucţiune va atribui variabilei Litere o valoare mulţime ai cărei membri sunt 'A' până la 'C' inclusiv.

Atunci când apare într-un constructor de mulţime, forma X..Y semnifică toate valorile cuprinse între X şi Y (inclusiv). În cazul în care X este mai mare decât Y, X..Y nu reprezintă nici o valoare. Nu este obligatoriu ca într-un constructor de mulţime să apară numai constante. De exemplu, se poate scrie:

MulţimeTest:=[N..N+6];

în care MulţimeTest este o variabilă mulţime, iar N este o variabilă de tip integer.
Un constructor de mulţime cu forma [] va reprezenta mulţimea vidă.

Diagrama de sintaxă a constructorilor de mulţime este prezentată în figura 12.22.

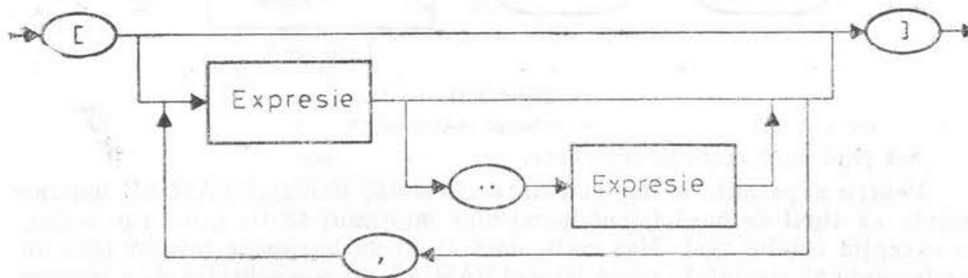


Fig. 12.22.

● **Operatorii** care se pot aplica asupra obiectelor cu structură mulțime sunt :

+ pentru reuniune;
* pentru intersecție;
— pentru diferență.

Cu ajutorul acestor operatori se pot construi expresii de tip mulțime. Asupra mulțimilor, privite ca operanzi, se pot aplica și operatorii:

= și <> pentru verificarea identității sau a neidentității a două mulțimi ;
<= și >= pentru verificarea incluziunii mulțimilor. Cu alte cuvinte, A<=B are înțelesul de „A este o submulțime a lui B" iar A>=B are înțelesul de „B este o submulțime a lui A";
in pentru verificarea apartenenței. Primul operand este de tip scalar iar cel de-al doilea este de tip mulțime; rezultatul are valoarea True numai atunci când primul operand este membru al celui de-al doilea.

Operațiile care folosesc mulțimi sunt destul de rapide și sunt utilizate de multe ori pentru eliminarea unor teste complicate. De exemplu, în loc de:
if(ch='a')**or**(ch='b')**or**(ch='c')**or**(ch='d')**or**(ch='z')**then** op;
unde ch este o variabilă de tip char, iar op este o instrucțiune, se poate folosi instrucțiunea:
if ch in ['a'..'d', 'z'] **then** op;

În limbajul PASCAL este permisă atribuirea de valori mulțime unor variabile de același tip. De asemenea, se pot folosi mulțimi ca parametri ai subprogramelor ; este evident însă că o funcție nu poate genera ca rezultat o mulțime.

● **Reguli ce se aplică în cazul mulțimilor:**

— Dacă operandul aflat în stânga operatorului **in** este de tip A, iar cel aflat în dreapta este de tip **set of B**, atunci A și B trebuie să fie tipuri identice sau să fie ambele subdomenii ale aceluiași tip sau să fie una subdomeniu al celeilalte ;
— Se spune că două tipuri mulțime sunt compatibile, dacă ele au același tip de bază sau dacă tipurile de bază sunt subdomenii ale aceluiași tip sau dacă unul dintre tipurile de bază este un subdomeniu al celuilalt;
— Două valori mulțime pot fi combinate folosind operatorii descriși anterior sau pot fi comparate numai dacă sunt de tipuri compatibile;
— O expresie de tip mulțime poate fi atribuită numai unei variabile mulțime de tip compatibil; fiecare membru al valorii expresiei mulțime trebuie să fie o valoare corespunzătoare ca tip cu tipul de bază al variabilei mulțime.

Exemplul 16. Generarea tuturor numerelor prime cuprinse în domeniul de valori 2..N, în care N>=2. Programul folosește un algoritm simplu, numit *„sita lui Eratostene"*.

1. Se pun în „sită" toate numerele cuprinse în domeniul 2 până la N;
2. Se alege și se extrage din „sită" cel mai mic număr ;
3. Se include acest număr în mulțimea numită „prim" ;
4. Se examinează „sita", extrăgând toți multiplii acestui număr ;
5. Dacă în „sită" mai există numere, se repetă pașii 2, 3, 4, 5.

```
program Sita1 (input, output);
const N=100;
var sita, prim:set of 2..N;
    next, j:integer;
```

```
begin {initializare}
   sita:=[2..N];
   prim:=[];
   next:=2;
   repeat {găseşte următorul număr prim}
    while not (next in sita) do next:=succ(next);
    prim:=prim+[next];
    j:=next;
    while j<=N do {eliminare}
     begin
       sita:=sita-[j];
       j:=j+next
     end
   until sita=[]
end.
```

Ca variantă, se poate scrie un program în care mulţimile să reprezin te numa numerele impare.

```
program Sita2 (input, output);
const N=50;
var sita, prim:set of 2..N;
    next, j,c:integer;
begin {iniţializare}
   sita:=[2..N];
   prim:=[];
   next:=2;
   repeat {găseşte următorul număr prim}
    while not (next in sita) do next:=succ(next);
    prim:=prim+[next];
    c:=2*next-1; {c=noul număr prim}
    j:=next;
    while j<=N do {eliminare}
     begin
       sita:=sita-[j];
       j:=j+c
     end
   until sita=[]
end.
```

EXERCIŢII

1. Care sunt diferenţele principale între tipurile de date array şi record ?

2. Sunt corecte următoarele declaraţii de variabile ? Motivaţi răspunsul.

```
var Contor:integer;
    Baza  :record
             Contor:real;
             Baza  :boolean
           end;
```

3. Să se definească tipul înregistrare conţinând: vârsta, numele, sexul, culoarea ochilor pentru un grup de persoane. Se vor declara apoi două variabile de acest tip.

4. Sunt corecte următoarele declaraţii de variabile ? Motivaţi răspunsul.

```
var Contor:integer;
    Baza :record
              Contor:real;
              Baza :boolean
          end;
```

5. Găsiţi şi corectaţi erorile din următoarea secvenţă de program:

```
type complex=record
              real:real;
              imaginar:real
          end;
var c1,c2:complex;
begin
    write ('C1=');
    readln(c1);
    write('C2=');
    readln(c2);
    c3. real:=c1, real+c2. real;
    c3. imaginar:=c1. imaginar+c2. imaginar;
    if c3. real:=0 then
        if c3. imaginar:=0 then
            writeln('Numărul este zero');
        else writeln ('Număr imaginar');
    else
        if c3. imaginar:=0 then
            writeln ('Număr real');
        else writeln (c3. real, '+', c3. imaginar)
end.
```

6. Ce erori conţine următorul program ?

```
program Greşit;
type rec=record
          x:integer;
          y:real
    var r:rec;
        f:file of rec;
        i:integer;
begin
    reset(f,'test.pas');
    read(f,r);
    writeln(r.x:5:2);
    close(f);
    read(f,r);
    writeln(r.y:5:2)
end.
```

7. Care sunt principalele deosebiri între fişiere şi tablouri ?

8. Scrieţi o funcţie care să întoarcă cel mai mare element al unei variabile de tip mulţime (cu elemente întregi).

9. Pentru o grupă de studenţi se introduc de pe mediul de intrare următoarele date
— numărul de studenţi din grupă ;
— numele şi prenumele fiecărui student ;
— numărul de examene din sesiunea de vară ;
— notele fiecărui student la examenele acestei sesiuni (0 pentru absenţă !).
Să se scrie programul care afişează, în ordinea mediilor, lista studenţilor care au cel mult o absenţă la examene (la egalitate de medie se respectă ordinea alfabetică).

10. Utilizând tipul de date înregistrare şi considerând un eşantion social de maximum 200 de persoane, să se efectueze calculele statistice în vederea afişării la terminalul de ieşire a următoarelor informaţii :
— procentul de persoane având vârsta sub 18 ani ;
— procentul de persoane având vârsta între 18 şi 62 de ani ;
— procentul de persoane având vârsta peste 62 de ani ;
— înălţimea medie a bărbaţilor între 18 şi 30 de ani ;
— procentul persoanelor căsătorite.

11. Definiţi un tip înregistrare convenabil pentru descrierea unei figuri geometrice plane. De exemplu, înregistrarea va conţine denumirea formei figurii geometrice şi :
— pentru un cerc — raza ;
— pentru un dreptunghi — dimensiunile celor două laturi.
Scrieţi apoi o funcţie care să aibă ca rezultat aria unei figuri geometrice date şi folosiţi-o într-un program.

12. Să se realizeze descrierea în PASCAL a informaţiilor relative la un grup de persoane (numărul maxim de persoane în grup este 1 000). Despre fiecare persoană se folosesc următoarele informaţii : nume, prenume, adresă, sex, greutate, înălţime, vârstă, culoare a părului şi a ochilor. Să se scrie un program care stochează informaţiile despre persoane pe disc magnetic într-un fişier care să poată fi citit apoi, dacă se doreşte, asigură tipărirea acestora într-o formă convenabilă. Programul se poate extinde astfel încât să solicite introducerea de la tastatură a numelui unei persoane şi să tipărească informaţiile despre toate persoanele cu numele respectiv (dacă există asemenea persoane). Dacă de la tastatură se introduce textul ORICARE, programul va tipări informaţiile despre toate persoanele, indiferent de nume.

13. Fie definiţia : **type** M=set of VMin..VMax; unde VMin şi VMax sunt constante întregi. Să se scrie procedura TipMult (x:M) care tipăreşte valoarea unei variabile de tip M. Ce se întâmplă în cazul în care VMin şi VMax sunt scalare oarecare (dar nu reale) ?

14. Să se definească un tip înregistrare care să permită declararea de variabile numere complexe şi cu ajutorul acestuia să se simuleze în limbajul Pascal toate operaţiile asupra numerelor complexe : adunarea, scăderea, înmulţirea, împărţirea, calcularea modulului, a argumentului, a părţii reale şi a părţii imaginare.

15. Să se scrie un program care, primind numele a trei fişiere formează fişierul obţinut prin concatenarea tuturor informaţiilor din acestea. Fişierele conţin numere reale.

CAPITOLUL 13

ALOCAREA DINAMICĂ A MEMORIEI

13.1. TIPUL INDICATOR (POINTER)

Să presupunem că se dorește crearea unei liste a numelor unor persoane, listă ordonată lexicografic. În acest scop, se poate folosi o structură de tip tablou în care să se insereze fiecare nume pe poziția corectă (în raport cu criteriul stabilit). Deoarece tabloul este o structură lipsită de flexibilitate, de fiecare dată când va fi inserat un nou nume, celelalte elemente ale tabloului vor trebui deplasate astfel încât să se creeze spațiul necesar pentru acesta. Cu cât este mai mare dimensiunea listei, cu atât numărul de elemente ce trebuie deplasate crește iar viteza de execuție a programului scade. (Necesitatea deplasării elementelor în tablou poate fi exemplificată astfel : dacă se dorește inserarea unui nou nume între al doilea și al treilea element, trebuie mutat cel de-al treilea element în cel de-al patrulea element, ceea ce presupune deplasarea celui de-al patrulea element în cel de-al cincilea și așa mai departe.)

O altă metodă ce poate fi folosită pentru rezolvarea acestei probleme este construirea unei liste, în cadrul căreia fiecare intrare să specifice în mod explicit intrarea următoare, ca în figura 13.1.

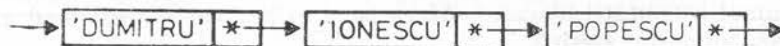


Fig. 13.1.

Referirile la intrările următoare din cadrul listei se numesc **indicatori (pointers)**. Avantajele folosirii unor astfel de liste sunt evidente.

● **Dacă** se **dorește inserarea** în listă a unui nou nume (de exemplu, 'ENESCU' între 'DUMITRU' și 'IONESCU') trebuie efectuate următoarele operații :

— crearea unei noi intrări care va conține noul nume ('ENESCU') și a unui indicator spre intrarea ce îl va urma ('IONESCU');

— redirectarea indicatorului asociat vechii intrări precedente ('DUMITRU') către intrarea nou creată ('ENESCU').

Observație : celelalte intrări componente ale listei rămân nemodificate (fig.13.2):

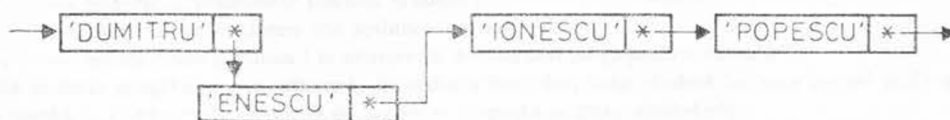


Fig. 13.2.

● **Dacă se dorește ștergerea** unui element al listei (de exemplu, ștergerea intrării asociate numelui 'IONESCU'), este necesară doar redirectarea indicatorului corespunzător intrării ce îl precede ('ENESCU') către intrarea ce îl urmează ('POPESCU'):

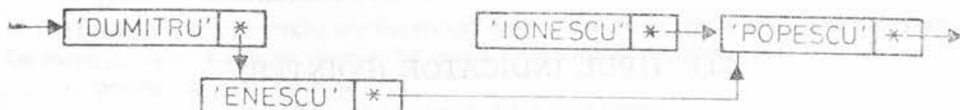


Fig. 13.3.

O listă în cadrul căreia *legătura între înregistrări este realizată cu ajutorul indicatorilor* se numește **listă înlănțuită**.

Indicatorii reprezintă date ce pot fi atribuite și comparate, folosirea lor permițând construirea listelor și a unor structuri de date complexe ce pot fi prelucrate prin simpla redirectare a indicatorilor, mărind astfel viteza de lucru în comparație cu alte metode ce ar putea fi utilizate.

În cadrul capitolelor precedente s-a arătat că în limbajul PASCAL variabilele se declară înainte de a fi utilizate, prin declarare asociindu-li-se un tip și un identificator. Tipul precizează atât structura cât și dimensiunea variabilei. Identificatorul reprezintă numele locației de memorie ce se alocă în mod automat variabilei respective în momentul declarării și care va fi rezervată pe toată durata execuției blocului de program din care face parte declarația. La ieșirea din bloc, locația este desființată. Cu alte cuvinte, zona de variabile a unui bloc se creează automat, în momentul activării lui, și există pe tot parcursul activității blocului. **Alocarea** și utilizarea în acest mod a variabilelor se numește **statică**.

În plus, limbajul PASCAL permite **alocarea** și eliberarea **dinamică** a unor zone de memorie, operații specificate în mod explicit de către programator. Spre deosebire de variabilele alocate static, accesibile prin numele lor, zonele alocate dinamic se caracterizează numai prin **adrese (indicatori)**, asociate în momentul alocării (și nu prin nume). Aceste zone se numesc **variabile alocate dinamic** deoarece *pot fi create și desființate în orice moment al execuției programului*.

Făcând o comparație între tablouri (structuri statice) și liste înlănțuite (structuri dinamice), se poate evidenția un alt mare avantaj al utilizării celor din urmă: dimensiunea unei liste înlănțuite poate fi extinsă sau redusă după dorință, în timp ce dimensiunea unui tablou trebuie precizată de la început de către programator, impunându-se astfel o limită asupra numărului de elemente ce pot fi memorate.

Exemplul 1. Să presupunem că se dorește crearea unui graf finit, adică o listă înlănţuită în care nodurile grafului să fie înregistrări cu câte două câmpuri, iar ramurile să fie indicatori. Pentru rezolvarea acestei probleme se pot scrie următoarele definiţii de tip:

```
type Referinţa=↑Noduri;
     Noduri   =record
       Element:Elemente;
       Urmator:Referinţa
       end;
```

(Explicitarea tipului Elemente nu este relevantă pentru exemplul prezentat.) Prima definiţie precizează faptul că Referinţa este un tip indicator (pointer) adică fiecare valoare a tipului Referinţă va fi adresa asociată unei variabile de tip Noduri şi care, la rândul ei, constituie o intrare aparţinând listei înlănţuite. Acesta este motivul pentru care cea de-a doua definiţie apare în forma prezentată, fiecare intrare conţinând nu numai un element ci şi un indicator către următoarea intrare a listei înlănţuite.

Accesarea fiecărei intrări a listei se face pornind de la intrarea ce o precede. Apare însă o problemă: în ce mod se pot marca şi în ce mod pot fi accesate prima şi respectiv ultima intrare?

Adresa sau indicatorul asociat primei intrări poate fi memorat într-o variabilă declarată astfel:

Antet : Referinţa ;

Pentru a marca ultima intrare a unei liste înlănţuite (deci pentru a arăta că după aceasta nu mai urmează nici o altă intrare), limbajul PASCAL furnizează o valoare indicator aparte, **nil** (cuvânt-cheie) care nu face referire la nici o locaţie. În cazul exemplului prezentat anterior, marcarea ultimei intrări a listei înlănţuite asociată grafului se va face prin memorarea valorii **nil** în câmpul Următor corespunzător, ca în figura 13.4:

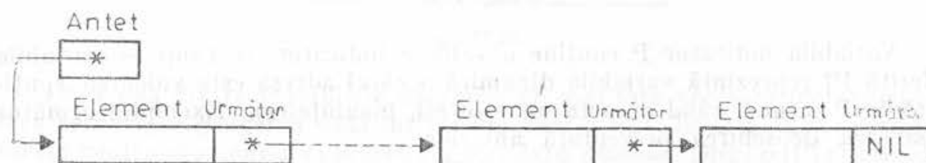


Fig. 13.4.

Observaţie: prin memorarea valorii **nil** în variabila Antet se poate reprezenta o **listă înlănţuită vidă**, adică o listă care *nu conţine nici o intrare.*

În limbajul PASCAL, valorile de tip indicator pot fi atribuite sau pot fi comparate cu ajutorul operatorilor relaţionali = şi <>. Ele pot fi folosite ca parametri ai subprogramelor şi pot constitui rezultate ale funcţiilor.

Observaţie: se spune că două valori indicator sunt egale atunci când ambele adresează aceeaşi variabilă dinamică sau atunci când ambele sunt **nil**.

Diagrama de sintaxă a tipului indicator folosit deja în cadrul Exemplului 1 este redată în figura 13.5, în care „Tip domeniu" reprezintă tipul varia-



Fig. 13.5.

bilelor dinamice adresate de către indicatori. El poate fi orice tip standard sau definit de către utilizator, dar, în mod uzual, se preferă folosirea tipului înregistrare.

Observații :

1. Este important de reținut faptul că „Tip domeniu" trebuie precizat cu ajutorul unui **identificator de tip** și nu în mod explicit ;
2. În cadrul unei definiții de tip indicator un identificator (de tip) poate fi referit înainte de a fi el însuși definit. Aceasta este unica excepție pe care o permite limbajul PASCAL privind utilizarea unui identificator în exteriorul domeniului său. Este obligatoriu însă ca definirea identificatorului astfel utilizat să se facă până la încheierea blocului.

Presupunând acum că P și Q sunt două variabile indicator de tip ↑T, cărora le-au fost atribuite valori (altele decât **nil**), variabilele dinamice adresate de P și Q se notează cu P↑ și, respectiv Q↑ și se numesc **variabile referite.**

Diagrama de sintaxă asociată unei variabile referite este prezentată în figura 13.6.

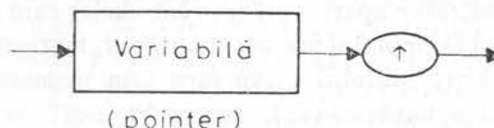


Fig. 13.6.

Variabila indicator P conține o valoare indicator, în timp ce variabila referită P↑ reprezintă variabila dinamică a cărei adresă este valoarea conținută în P (atunci când aceasta nu este **nil**, bineînțeles). Exemplul următor ilustrează deosebirea prezentată anterior:

Exemplul 2. Presupunem că X și Y sunt două valori de tip T și că există situația ilustrată în figura 13.7.

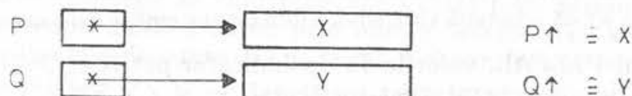


Fig. 13.7.

Instrucțiunea P:=Q atribuie o valoare indicator și, ca urmare, în urma executării acestei instrucțiuni, P și Q vor adresa aceeași variabilă dinamică (fig. 13.8).

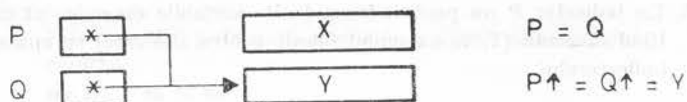


Fig. 13.8.

Dacă însă, din situaţia iniţială, se execută instrucţiunea P↑:=Q↑, variabilei dinamice adresate de P îi va fi atribuită valoarea variabilei dinamice adresate de Q, de data aceasta P şi Q păstrându-şi valorile iniţiale, ca în figura 13.9.

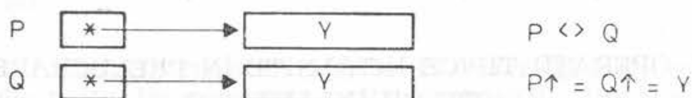


Fig. 13.9.

Observaţie : scrierea unor instrucţiuni cum sunt, de exemplu, P↑:=Q sau P:=Q↑ nu este permisă deoarece nu are nici un înţeles, încălcând regulile de atribuire cu referire la tip (P şi Q sunt adrese (indicatori) iar P↑ şi Q↑ sunt variabile dinamice, adică locaţii de memorie).

În limbajul PASCAL standard, declararea unei variabile indicator, (P:↑T;) creează numai locaţia asociată lui P, valoarea iniţială a acestuia rămânând nedefinită şi urmând a i se atribui ulterior (ca oricărei alte variabile) o valoare indicator sau valoarea **nil**.

Crearea variabilelor dinamice se realizează cu ajutorul procedurii standard **New**. De exemplu, New(P) creează o nouă variabilă dinamică de tip T, iniţial nedefinită, şi face ca aceasta să fie adresată de către P (fig. 13.10).



Fig. 13.10.

Dacă una dintre variabilele dinamice nu mai este necesară (de exemplu, dacă se doreşte ştergerea unei intrări din cadrul unei liste înlănţuite) se foloseşte pentru eliminarea ei procedura standard **Dispose**. Dispose(P) eliberează locaţia de memorie ocupată de variabila dinamică P↑ (aceasta putând fi utilizată ulterior în alt scop) şi lasă indicatorul P nedefinit.

Observaţii :

1. Redirectarea indicatorilor în vederea ştergerii unei intrări din lista înlănţuită trebuie programată în mod explicit. Ea nu este realizată de către procedura Dispose;
2. New(P) realizează crearea unei variabile dinamice. În limbajul PASCAL nu există posibilitatea ca o variabilă indicator să adreseze o variabilă statică (ce are asociat un nume) astfel încât situaţia din figura 13.11 nu poate să apară;



Fig. 13.11.

3. Un indicator P nu poate adresa decât variabile dinamice al căror tip este tipul domeniu (T, în exemplul folosit pentru ilustrare) ce apare în definiția indicatorului.

● **Reguli ce trebuie respectate.**

Atribuiri: — o valoare pointer poate fi atribuită numai unei variabile pointer de același tip;
— valoarea **nil** poate fi atribuită oricărei variabile pointer;

Comparații: — se pot compara între ele oricare două valori indicator de același tip;
— se poate compara oricare valoare pointer cu valoarea **nil**.

13.2. OPERAȚII TIPICE ÎNTÂLNITE ÎN PRELUCRAREA LISTELOR ÎNLANȚUITE

Dacă se dorește folosirea sistemului de calcul în scopul păstrării evidenței persoanelor care lucrează în cadrul unei întreprinderi, se poate scrie un program în care să se definească următoarele tipuri:

```
type Legătura=↑Persoana;
     Persoana=record
               Nume:     packed array [1..20] of char;
               Prenume:  packed array [1..20] of char;
               Marca:    integer;
               Urmator:  Legătura
             end;
```

Desigur, în cadrul tipului înregistrare Persoana pot să apară și alte câmpuri, după dorință, dar, în cazul luat în discuție, este suficientă o formă chiar mai simplă decît cea prezentată și anume:

```
type Persoana=record
              Marca:    integer;
              Următor:  Legătura;
            end;
```

unde Marca reprezintă numărul (de legitimație, de exemplu) asociat fiecărei persoane în cadrul evidenței.

Exemplul 3. Crearea listei inlănțuite, presupunând că în intreprindere există N persoane, va avea în final forma din figura 13.12.



Fig. 13.12.

Fragmentul de program cu ajutorul căruia se realizează crearea unei astfel de liste este:

```
var Primul, P:Legătura;
    i, M, N: integer;
```

```
begin
    Primul:=nil; {lista va fi creată de la sfârşit către început}
    read(N);
    for i:=1 to N do
      begin
        read(M);
        new(P);
        P↑.Următor:=Primul;
        P↑.Marca:=M;
        Primul:=P
      end
end;
```

În cadrul acestui fragment de program P↑ reprezintă însăşi înregistrarea (o intrare din cadrul listei) în timp ce P↑. Marca şi P↑. Următor reprezintă câmpurile cu numele Marca şi, respectiv, Următor ce o compun.

Exemplul 4. Prelucrarea secvenţială a listei înlănţuite. Presupunem că numărul total de intrări conţinute în listă nu este cunoscut şi că se doreşte calcularea acestui număr. În acest scop se poate folosi următoarea funcţie :

```
function NrIntrări (Primul:Legătura):integer;
  var Contor:integer;
      Crt:Legătura;
  begin
      Contor:=0;
      Crt:=Primul;
      while Crt<> nil do
        begin
          Contor:=Contor+1;
          Crt:=Crt↑.Următor
        end;
      NrIntrări:=Contor
  end;
```

în care variabila Crt reprezintă, la fiecare iteraţie, indicatorul către intrarea curentă.

Observaţie : Deoarece Crt↑. Urmator reprezintă câmpul Urmator al înregistrării (intrării) Crt↑, executarea instrucţiunii Crt:=Crt↑. Următor are drept consecinţă avansarea indicatorului Crt la următoarea intrare a listei.

Exemplul 5. Căutare liniară în listă. Se doreşte găsirea şi examinarea intrării al cărui câmp Marca are valoarea 40. În acest scop se introduc în program instrucţiunile:

```
Crt:=Primul;
while Crt↑.Marca <> 40 do Crt:=Crt↑.Următor;
```

Dacă în cadrul listei există cel puţin o intrare al cărui câmp Marca are valoarea 40, programul va funcţiona corect, altfel însă apare o situaţie nedorită, şi anume o buclă infinită. Pentru a elimina acest neajuns, se preferă folosirea următoarei variante :

```
Crt:=Primul;
while (Crt <> nil) and (Crt↑. Marca <> 40) do Crt:=Crt↑.Următor
```

O altă variantă de program care rezolvă aceeași problemă într-un mod elegant și corect este:

```
var B:boolean;
Crt:=Primul;
B:=true;
while (Crt <> nil) and B do if (Crt↑. Marca=40) then B:=false
                            else Crt:=Crt:↑.Următor;
```

în care variabila B poate fi folosită în continuare în cadrul programului pentru a semnala încheierea cu succes sau nu a operației de căutare liniară în listă a elementului dorit.

Exemplul 6. Inserarea în listă a unei noi intrări (numită Nou). Înainte de realizarea operației dorite, lista are forma din figura 13.13.

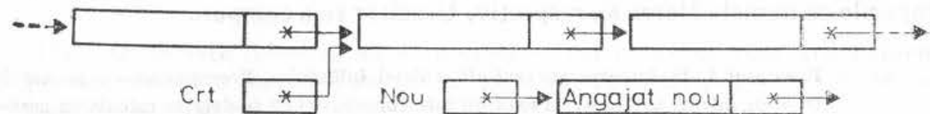


Fig. 13.13.

În program trebuie să apară următoarele declarații și instrucțiuni:

```
var Nou:Legătura;
begin
  ...
new(Nou);
read(Nou↑.Marca);
  Nou↑.Următor:=Crt↑.Următor;
  Crt↑.Următor:=Nou;
```

Observație: Scrierea instrucțiunilor de mai sus este justificată prin faptul că inserarea unei noi intrări în lista înlănțuită înseamnă de fapt redirectarea unor indicatori.

În urma executării acestui fragment de program, lista va avea forma din figura 13.14.

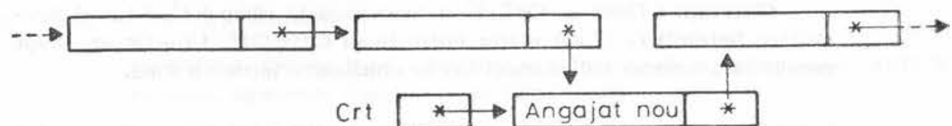


Fig. 13.14.

Exemplul 7. Ștergerea unei intrări din cadrul listei.

```
var Anterior:Legătura;
begin
  ...
  {se șterge intrarea curentă}
  Anterior↑.Următor:=Crt↑.Următor;
  Dispose (Crt);
  ...
```

Observaţie : Eliberarea locaţiei, realizată prin apelarea procedurii Dispose trebuie să se efectueze **numai după** redirectarea indicatorilor, prin care se „taie" legăturile intrării cu restul listei.

EXERCIŢII

Notă :

În cadrul exerciţiilor de programare de la acest capitol, implicând utilizarea listelor unidirecţionale sau circulare, se vor folosi tipurile:

```
type list =↑cel;
     cel=record
              k: integer;
              next: list
         end;
```

Componenta k a elementelor din listă poartă şi denumirea de „*cheie*".

Dacă se consideră:

l — pointer spre prima celulă şi

lf — pointer spre ultima celulă

atunci :

a) când lista este simplă, lf.next=nil

b) când lista este circulară, atunci lf.next=l

Pentru listele unidirecţionale, se consideră definită următoarea funcţie „constructor" (cons) :

```
function cons(k:integer; next:list):list;
var r: list;
begin
     new(r);
     r↑.k:=k;
     r↑.next:=next;
     cons:=r
end;
```

1. Să se scrie o funcţie care, primind o listă l, calculează suma tuturor elementelor pozitive componente.

2. Să se scrie o procedură care tipăreşte numărul de elemente pare, impare şi respectiv pozitive dintr-o listă.

3. Să se scrie o funcţie care calculează numărul de elemente impare dintr-o listă circulară.

4. Să se scrie o funcţie care, primind 2 liste l1 şi l2, decide dacă cele 2 sunt egale.

5. Să se scrie o funcţie „*member*(l, k)" care, primind o listă l şi o cheie k, decide dacă cheia k apare in lista l sau nu.

6. Să se reprezinte mulţimi finite prin intermediul listelor. Să se scrie funcţiile care calculează :

a) intersecţia a două mulţimi;

b) diferenţa a două mulţimi;

c) reuniunea a două mulţimi.

7. Să se insereze o nouă cheie k intr-o listă ordonată crescător.

8. Să se scrie o funcţie care, primind o listă l, construieşte o nouă listă r cu aceleaşi chei ca şi l, dar ordonate crescător.

9. Să se utilizeze listele pentru reprezentarea numerelor întregi, arbitrar de mari, prin şirul cifrelor lor, în baza 10. De exemplu, numărul 31706 este reprezentat de lista din fig. 13.15.

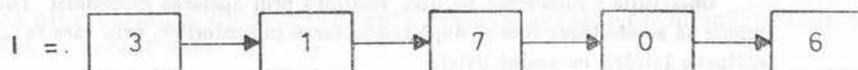


Fig. 13.15.

Să se scrie o funcţie care, primind un număr întreg x, calculează şi întoarce drept rezultat lista ce reprezintă numărul respectiv ca număr mare.

10. Să se scrie o funcţie care, primind 2 numere mari, calculează suma lor (reprezentată tot printr-o listă). Astfel, dacă cele 2 numere sunt x1 = 9568 şi x2 = 504, deci cele 2 liste sunt (fig. 13.16)

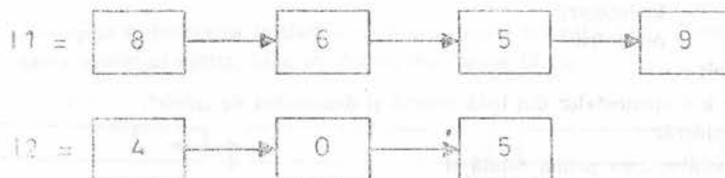


Fig. 13.16

atunci suma este x1 + x2 = 10072, iar funcţia va întoarce drept rezultat lista:

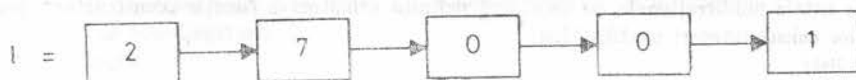


Fig. 13.17.

11. Să se scrie o funcţie pentru efectuarea diferenţei a 2 numere mari (v. problema 10).

12. Să se scrie o funcţie pentru efectuarea produsului a 2 numere mari (v. problema 10).

13. Să se reprezinte polinoamele prin intermediul listelor având următoarea structură:

```
type pol=↑cel;
     cel=record
          c:real; {coeficientul}
          g:integer; {gradul}
          next:pol
        end;
```

Lista trebuie să fie ordonată crescător după gradul monoamelor. Astfel polinomul:

$$P(x) = 9x^3 + x^5 - 4x^2 + 8$$

va fi reprezentat de lista din fig. 13.18.



Fig. 13.18.

Monoamele având coeficientul 0 sunt eliminate.

Să se scrie o funcţie care, primind un polinom p şi o valoare reală x, calculează valoarea reală p(x) efectuând un număr minim de înmulţiri.

14. Să se scrie o funcţie pentru calculul sumei a două polinoame (v. problema 13).

15. Să se scrie o funcţie pentru calculul produsului a două polinoame (v. problema 14).

CAPITOLUL 14

INSTRUCȚIUNEA GOTO (SALT NECONDIȚIONAT)

Marea majoritate a problemelor de programare se poate rezolva excelent utilizând un număr limitat de structuri de bază : secvența, selecția (prin instrucțiunile **if** sau **case**), repetiția (**while**, **repeat** sau **for**) și, desigur, subprogramele (proceduri și funcții).

Însă, ca excepții, întâlnim probleme în care fluxul de control intrinsec al algoritmului de rezolvare se prezintă excesiv de neregulat pentru a putea fi exprimat într-un limbaj de programare apelând exclusiv la instrucțiuni de atribuire, selective, repetitive sau de apel.

O situație în care structurile de mai sus nu sunt eficiente este cea în care, în cursul operațiilor de intrare, detectarea unei valori incorecte compromite desfășurarea ulterioară a prelucrărilor de date și, prin urmare, este obligatorie terminarea forțată a operațiilor de intrare cât și, eventual, a întregului program (semnalându-se faptul printr-un mesaj adecvat). Exemplul de mai jos corespunde acestei situații.

Exemplul 1. Se cere scrierea unei proceduri pentru citirea unei matrici pătrate de dimensiune N × N, cu elementele numere pozitive. Introducerea elementelor matricei se termină prin tastarea unei valori negative. Analizând problema, constatăm posibilitatea unor variante nedorite :

— fie numărul de elemente este insuficient (mai mic decât N^2) ;

— fie se introduc prea multe elemente (mai mult de N^2), adică se introduc în continuare numere pozitive, dincolo de cel de-al N la puterea a doua element.

Problema se rezolvă prin verificarea fiecărui element introdus și, în cele două variante de mai sus, prin terminarea forțată a procedurii, nu înainte de a fi semnalată, printr-un mesaj de eroare, această terminare forțată.

Se consideră că a fost definit în programul principal tipul :

```
Matrice=array [1..N, 1..N] of 0..Max Int ;
```

iar N este o constantă aprioric definită.

```
procedure CitMat (var M:Matrice) ;
  label 100 ;
  var
```

```
      Element:integer;
      i, j:1..N;
   begin
      for i:=1 to N do
         for j:=1 to N do
         begin
            read (Element);
            if Element >=0 then M[i, j]:=Element
            else
               begin
               writeln('Prea puține elemente');
               goto 100
               end;
         end;
         read (Element); {ar trebui să fie un număr negativ}
         if Element >=0 then writeln ('Prea multe elemente');
      100:
      end;
```

● Exemplul ilustrează **modul de utilizare a instrucțiunii goto** :

— instrucțiunea **goto** asigură saltul în program, **necondiționat**, la o instrucțiune căreia i se atașează o **etichetă**. Aceasta trebuie declarată printr-o declarație de etichetă (precedată de cuvântul rezervat **label**), secțiunea etichetelor fiind prima dintre secțiunile părții declarative a programului (înainte de secțiunea constantelor, de cea a variabilelor etc.) ;

— eticheta urmată de ':' (de exemplu, 100:), se atașează instrucțiunii imediat următoare (în procedura precedentă atașarea etichetei se face la o instrucțiune fictivă (vidă) întrucât imediat urmează delimitatorul **end** al procedurii) ;

— o etichetă PASCAL are între unul și patru caractere tip cifră (de exemplu 10, 999, 4567 etc.). Variantele Turbo-PASCAL permit folosirea ca etichete și a identificatorilor (de exemplu Et02, Intrare1 etc.) ;

— etichetele folosite în cadrul unui bloc trebuie să fie distincte ;

— efectul unei instrucțiuni **goto** constă în trecerea controlului execuției programului la instrucțiunea etichetată referită, abandonându-se orice prelucrare existentă între aceste două instrucțiuni (în exemplul precedent, iterațiile buclei **for** sunt abandonate și nici instrucțiunile Read (Element) și **if** următoare nu mai sunt executate).

● Se poate eticheta orice instrucțiune, însă există câteva **restricții privind plasarea unei instrucțiuni goto** în raport cu instrucțiunea referită de aceasta :

— nici o instrucțiune **goto** nu trebuie să forțeze un salt din exterior în interiorul unei instrucțiuni structurate (compusă, **if, case, while, repeat, for** sau **with**). De exemplu, scrierea de mai jos este **incorectă** :

```
   for...do
      begin
         ...
         1:...
         ...
      end;
   goto 1;
```

— nici un **goto** nu are dreptul să forțeze un salt din exterior în interiorul unui subprogram (rezultatele devin ambigue, imprevizibile, ca și în cazul precedent);

• Desigur, se pot formula imediat **câteva observații.** Astfel, în exemplul prezentat, procedura ar putea fi rescrisă utilizându-se o variabilă suplimentară (de tip fanion (Flag)), returnată de procedură pentru a specifica, de exemplu :

— introducerea corectă de date (Flag are valoarea true) ;
— introducere incorectă de date (Flag are valoarea false).

Sau, pentru o mai exactă descriere a situațiilor incorecte, considerând Flag ca variabilă de tip integer :

— introducere corectă de date (Flag are valoarea 0) ;
— introducere incorectă de date, prea puține elemente (Flag are valoarea 1) ;
— introducere incorectă de date, prea multe elemente (Flag are valoarea 2).

Rescrisă, procedura are următoarea formă :

```
procedure CitMat (var M:Matrice; var Flag:integer);
  var
    Element; integer;
    i,j:1..N+1;
  begin
    Flag:=0;
    i:=1;
    j:=1 ;
    while (Flag=0) and (i<=N) do
      begin
        while (Flag=0) and (j<=N) do
          begin
            read (Element) ;
            if Element >=0 then
            begin
              M[i, j]:=Element;
              j:=j+1
            end
            else Flag:=1
          end;
        j:=1 ;
        i:=i+1
      end;
    if (Flag=0) then
    begin
      read(Element);
      if (Element >=0) then Flag:=2
    end
  end;
```

O consecință a teoremei de structură (Boehm și Jacopini) este următoarea : „Orice program se poate transforma într-un program structurat prin utilizarea funcțiilor și deciziilor programului inițial și prin introducerea și testarea unei variabile suplimentare“, rescrierea de mai sus a procedurii ilustrând această afirmație.

● **O altă observaţie** care se impune este aceea că, pentru cazul concret dat, era posibilă şi o altă modalitate de operare, verificând „local" datele, după exemplul de mai jos :

1. Până la elementul N la puterea a doua, inclusiv :

```
repeat
    write('Precizaţi elementul',i:2,',',j:2,' :');
    readln(M[i,j])
until (M[i,j]>=0) and (M[i,j]=trunc(M[i,j]))
```

validare de date care verifică şi faptul că numărul introdus este întreg şi nu real. În plus, dispare necesitatea existenţei variabilei Element ;

2. Pentru elementul N^2+1 (terminatorul introducerii de date, practic redundant în cazul utilizării strategiei 1.), se pot scrie instrucţiunile :

```
repeat
    write('Introduceţi terminatorul de date (număr negativ)');
    readln(Element)
until (Element <0);
```

Dacă s-a prevăzut condiţionarea 1, aceste instrucţiuni devin inutile. Iată două exemple pentru care s-au scris instrucţiunile în două variante : cu şi fără goto.

Exemplul 2.

● *Varianta cu goto :*

```
if X > Y then goto 1;
Max:=Y;
goto 2;
1:Max:=X;
2:write(Max);
```

● *Varianta fără goto :*

```
if X <= Y then Max:=Y
else Max:=X;
write(Max);
```

Exemplul 3.

● *Varianta cu goto :*

```
1:if R < N then goto 2;
R:=R—N;
goto 1;
2:write(R);
```

● *Varianta fără goto :*

```
while R >=N do R:=R—N;
write(R);
```

Orice program poate fi scris folosind numai **if**... **then**... şi **goto**, adică se poate renunţa la folosirea instrucţiunilor **while**, **repeat**, **for**, **case**. Însă un astfel de program ar fi foarte greu de înţeles : nu este uşor de înţeles nici chiar efectul unei singure instrucţiuni **goto** în fluxul controlului execuţiei, fără a ne uita la instrucţiunea etichetată la care se referă şi care se poate afla la depărtare de câteva pagini de instrucţiunea **goto**. În cadrul unui program care conţine multe instrucţiuni **goto**, fluxul controlului este foarte confuz. Astfel de programe mai sunt numite şi *„programe spaghetti"*.

De exemplu, organizarea din figura 14.1 devine practic ininteligibilă datorită utilizării excesive a instrucţiunilor **goto** (în figură, acestor instrucţiuni le corespund ramificaţiile ce conduc la nodurile conectoare 1, 2 şi 3).

Principiile programării structurate recomandă evitarea folosirii instrucţiunii **goto**, mulţi programatori neutilizând-o deloc. Există totuşi situaţii în care regula eliminării instrucţiunilor **goto** nu trebuie aplicată dogmatic : în

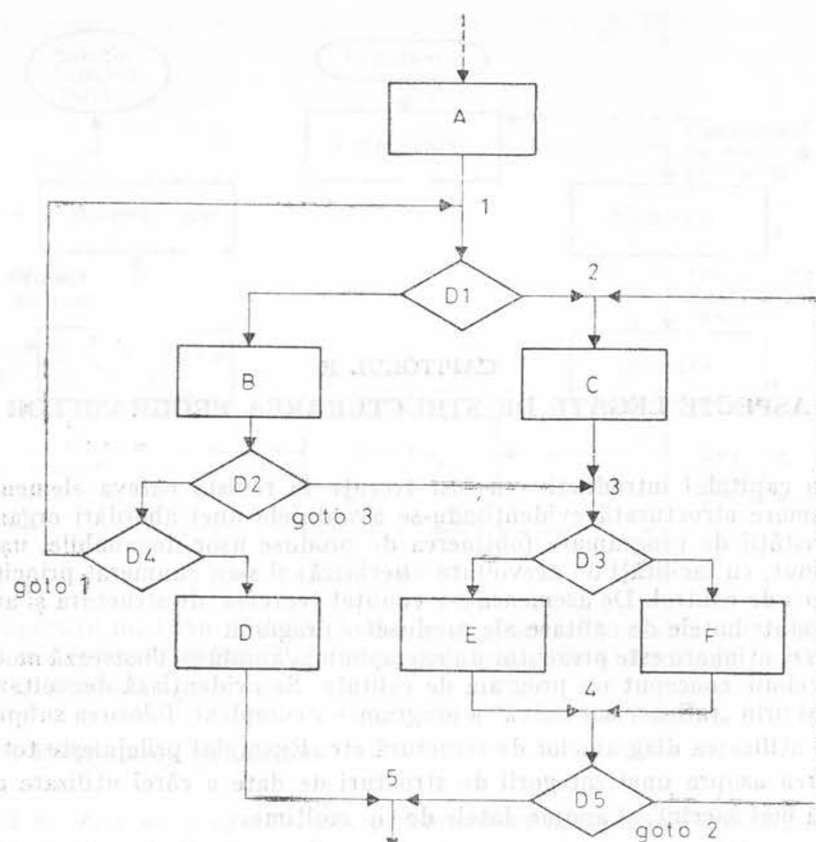


Fig. 14.1.

fond, scopul unei programări de calitate este acela de a produce programe eficiente, ușor de citit și de înțeles. Cele mai frecvente situații sunt cele în care apare necesitatea ieșirii forțate dintr-o buclă sau dintr-un subprogram (mai ales în cazul saltului „înainte" la care structurarea programului devine foarte anevoioasă).

CAPITOLUL 15

ASPECTE LEGATE DE STRUCTURAREA PROGRAMELOR

În capitolul introductiv au fost trecute în revistă câteva elemente de programare structurată evidențiindu-se avantajele unei abordări organizate a activității de programare (obținerea de produse ușor depanabile, ușor de întreținut, cu facilități de dezvoltare ulterioară) și s-au enumerat principalele structuri de control. De asemenea s-a enunțat teorema de structură și au fost definite atributele de calitate ale produselor program.

În continuare este prezentat un exemplu mai amplu ce ilustrează modul în care trebuie conceput un program de calitate. Se evidențiază dezvoltarea în etape și prin „rafinare succesivă" a programelor complexe, folosirea subprogramelor, utilizarea diagramelor de structură etc. Exemplul prilejuiește totodată revenirea asupra unei categorii de structuri de date a cărei utilizare a fost tratată mai succint, și anume datele de tip mulțime.

15.1. ETAPELE GENERALE DE REZOLVARE A UNEI PROBLEME. PUNCTE DE REPER ÎN REALIZAREA UNUI PROGRAM

În general, orice problemă tehnică, indiferent de natura sa, parcurge până la materializarea soluției sale, următoarele etape, schematic reprezentate în figura 15.1. Procesul de rezolvare reprezentat în figura 15.1 nu este unul liniar : iterațiile și revenirile îi sunt specifice. Astfel, din orice etapă se poate reveni la oricare dintre etapele precedente (fapt sugerat prin săgețile întrerupte ale schemei din figură), oricând putându-se constata o disfuncționalitate sau lipsa unor elemente produse în faze anterioare etapei curente, neplăceri rezolvabile numai prin reluarea unor etape deja parcurse.

Nu de puține ori, chiar în ultima etapă, cea de implementare, se poate constata o eroare majoră în proiectare sau chiar în formularea detaliată a problemei, eroare ce face imposibilă sau nesatisfăcătoare aplicarea soluției finale. Particularizând elementele de mai sus pentru activitatea de elaborare a programelor, se constată regăsirea tuturor fazelor din desenul inițial. În continuare se insistă numai asupra unor particularități strict necesare în cazul

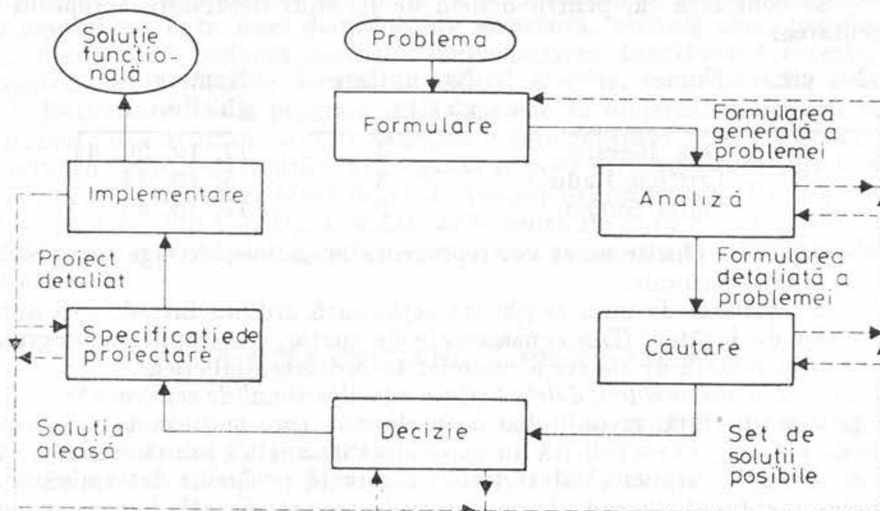


Fig. 15.1.

unor aplicații mici sau mijlocii. Astfel, este „jalonată" rezolvarea problemelor pe un caz concret, lăsând cititorului posibilitatea de a desprinde concluzii cu caracter general.

15.1.1. Formularea problemei

Să se scrie un program care, acceptând ca date de intrare numărul de jucători la un concurs de tip Prono-Expres, numele acestora și numerele jucate de fiecare în parte, furnizează, pe baza numerelor extrase într-o etapă de joc, lista concurenților în ordinea numărului de numere ghicite.

15.1.2. Analiza problemei

În această etapă se urmărește eliminarea oricăror ambiguități sau neclarități ale formulării generale. Un bun principiu este acela de a stabili foarte exact ce date de intrare se furnizează, ce date de ieșire se dorește a fi obținute și cum trebuie prelucrate intrările pentru a obține rezultatele solicitate. Desigur, experiența proiectantului de programe joacă un rol important în minimizarea efortului de parcurgere a fazelor. Vor fi luate în considerare doar câteva dintre chestiunile pe care și le pune acesta pentru a defini corect problema, numărul acestor chestiuni nefiind însă limitat.

* *Ce precizări trebuie făcute asupra datelor de intrare ?*

Pentru aplicația de față se consideră că :

— numărul de jucători poate fi cel mult 200 ;
— gama numerelor jucate este 1..45;
— numele și prenumele unui jucător pot avea împreună maximum 25 de caractere;
— fiecare jucător propune o singură variantă de joc, cu 5 numere;
— la o etapă de joc se extrag 12 numere.

Ce precizări suplimentare sunt necesare referitor la datele de ieșire ?

— se consideră că, pentru o listă de jucători dată, este acceptată reprezentarea:

Nr. crt.	Nume	Nr. numere ghicite	Numere ghicite
1	Dan Ionescu	4	17, 19, 20, 44
2	Cristian Radu	3	5, 19, 44
3	Maria Stoian	3	17, 19, 45

— numerele ghicite nu se vor reprezenta în ordinea extragerii ci potrivit ordinii naturale;
— la egalitate de numere ghicite acționează ordinea introducerii numelor de jucător. (Din considerente de spațiu, nu complicăm programul prin cerință de listare a numelor în ordinea alfabetică).

* *În ce mod se vor prelucra datele ? (Care este algoritmul de rezolvare ?)*

În cazul de față, modalitatea de prelucrare (sau mai exact spus necesitățile de prelucrare) nu solicită un mare efort de analiză așa cum s-ar întâmpla, de exemplu, atunci când ar trebui rezolvată problema determinării extremului unei funcții în cadrul unui interval precizat, situație în care ar putea fi formulate mai multe variante. În cazul studiat este nevoie doar să verificăm pentru fiecare jucător în parte, dacă numerele jucate sunt sau nu identice cu cele extrase și să contorizăm numărul de coincidențe.

* *Este nevoie ca programul să îndeplinească funcții suplimentare ?*

Desigur, orice aplicație poate fi lărgită, perfecționată sau redimensionată. În cazul prezentat, ne vom limita însă la funcțiile specificate în formularea inițială a problemei.

15.1.3. Alegerea soluției

Iată câteva dintre posibilitățile de rezolvare a problemei. Problema în sine conține mai puțin calcule (aritmetice sau logice) și mai mult reprezentări ale datelor. Fiind vorba despre un volum relativ mare al datelor de intrare, cu semnificații diverse, se poate opta pentru folosirea tablourilor, a fișierelor, a mulțimilor sau a listelor înlănțuite, fiind aproape evident că între tipurile de date ce vor fi folosite trebuie să figureze și înregistrările care să conțină informațiile de intrare aferente fiecărui concurent (nume, numere jucate). Însă, pentru a examina încă un exemplu de problemă rezolvată cu ajutorul datelor de tip mulțime, seturile de numere (jucate, extrase, ghicite) vor fi reprezentate cu ajutorul acestora. Evident, în acest caz, determinarea numărului de numere ghicite și specificarea acestora se va realiza cu multă ușurință, efectuând o simplă intersecție de mulțimi și calculând cardinalul acesteia.

15.1.4. Specificația de proiectare și implementarea

Pentru tratarea acestor faze (comasate aici) se recomandă următoarea metodologie:

— pe baza operațiilor (funcțiilor) solicitate programului, se elaborează o **primă schiță** a acestuia (un nucleu al dezvoltării modulare);
— se „rafinează" în câteva **iterații succesive** structura programului, gradul de detaliere crescând la fiecare iterație.

În general, **implementarea** constă în scrierea instrucţiunilor (codificarea programului pe baza unei diagrame de structură, organigrame, pseudocod etc.), însoţită de testarea acestora (demonstrarea funcţionării corecte) şi depanarea lor (depistarea locului şi naturii erorilor, cu corectarea acestor erori). Dezvoltarea unui program nu se încheie în momentul obţinerii sale. Programul intră acum în faza de **întreţinere** care continuă pe toată durata sa de viaţă. În timp, asupra oricărui program se pot opera modificări, fie în scopul corectării unor erori identificate în timpul utilizării sale, fie pentru a-l adapta unor cerinţe noi. Dacă modificările solicitate sunt majore, este necesară reluarea activităţilor specifice etapei de analiză sau chiar reformularea problemei.

15.2. SCHIŢA INIŢIALĂ A PROGRAMULUI

Se observă că programul trebuie să conţină *instrucţiuni* care să **asigure :**

— citirea numărului de jucători ;
— citirea numelui fiecărui jucător şi a numerelor jucate de acesta ;
— citirea numerelor extrase ;
— prelucrarea numerelor (găsirea numărului de numere ghicite) ;
— ordonarea listei ;
— afişarea listei ordonate.

Toate acestea determină construirea unei **diagrame de structură** (ierarhie de module-program) **clasică**, de tipul IPO (Input-Process-Output (Intrare-Prelucrare-Ieşire)) (fig. 15.2), diagramă ce poate fi rafinată ca în figura 15.3.

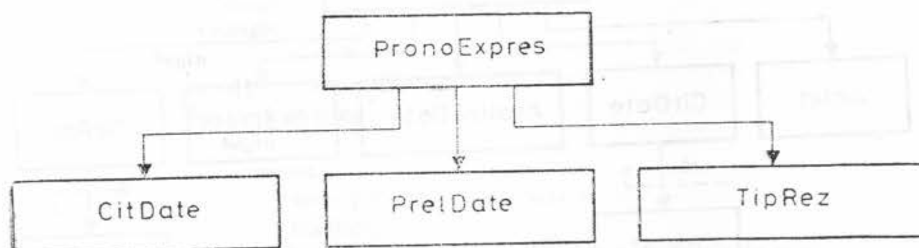


Fig. 15.2.

Analizând structura prezentată anterior, se constată existenţa unor operaţii mai simple (Citeşte număr jucători, Tipăreşte nume jucător) şi a unora mai complexe (Citeşte numere jucate, Tipăreşte numere ghicite), acestea fiind operaţii de intrare/ieşire în care se vor folosi mulţimi. Nu este necesar ca structura să fie modularizată prea mult, operaţiile simple putând fi incluse în nivelul superior (coordonator). Întrucât „citirea urnei" este o operaţie similară „citirii numerelor jucate de un jucător", se va folosi un modul unic ce va servi ambeleor scopuri : CitMult — citeşte mulţime.

De asemenea, „tipărirea numerelor ghicite" de către fiecare jucător se va transforma într-o procedură de afişare a elementelor unei mulţimi: Scrie Mult.

În ceea ce priveşte modulul PrelDate, el nu are decât rolul de a apela repetat un submodul de „analiză numere jucător" (Determinarea numărului de numere ghicite/jucător) şi o singură dată „ordonează lista" fiind posibilă reducerea unui nivel intermediar, pentru ca, în final, să rezulte diagrama de structură din figura 15.4.

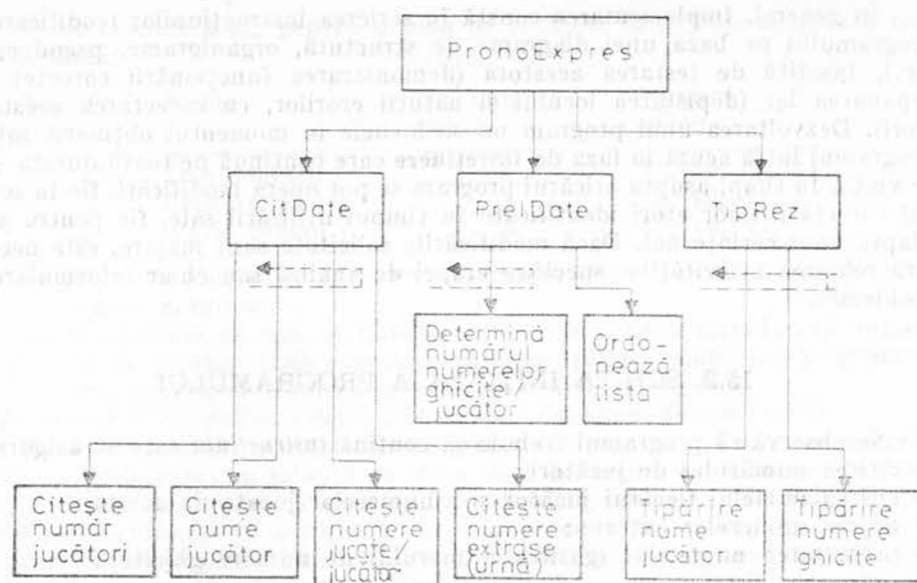


Fig. 15.3.

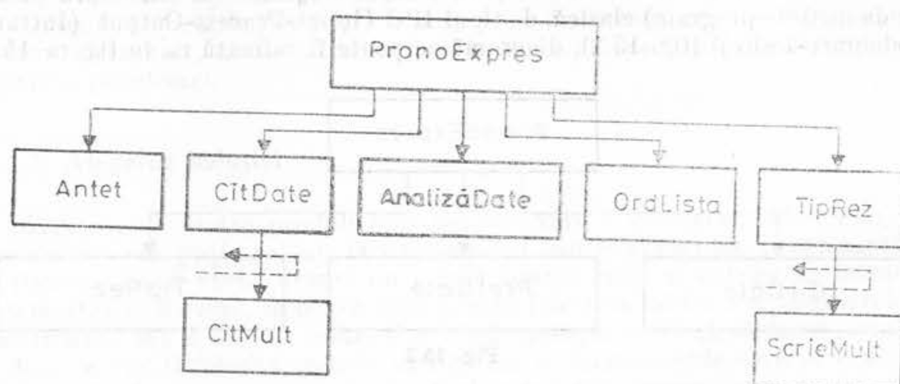


Fig. 15.4.

În cadrul programului a fost prevăzut şi un modul de iniţializare, care apare, în general, în majoritatea programelor; în cazul de faţă el are rolul de a trimite un mesaj de „prezentare", şi de aceea a fost numit Antet.

Considerând următoarele condiţii de reprezentare a datelor:

* Constante:

```
Maxim      = 45; {cel mai mare număr din set}
NrMaxJuc=200; {numărul maxim de jucători}
NrExtrase= 12;
NrPropuse=  5;
```

* Tipuri de date:

```
Numere    =1..Maxim;
Mulţime   =set of Numere;
Sir       =packed array [1..25] of char;
Concurent=record
```

```
        Nume: Sir;
        MNrJuc,MNrOK: Mulţime;
        {MNrJuc reprezintă mulţimea numerelor jucate}
        {MNrOK reprezintă mulţimea numerelor ghicite}
        nng: 0.. NrPropuse;
        {nng reprezintă numărul de numere ghicite}
      end;
   Registru  =array [1..NrMaxJuc] of Concurent;
```

* Variabile:

```
   Lista:Registru; {lista tuturor concurenţilor}
   Urna: Mulţime; {mulţimea numerelor extrase}
```

Sunt descrise mai întâi procedurile de lucru elementare (CitNume), cele de lucru cu mulţimi (CitMult, ScrieMult, Cardinal) şi apoi procedurile mai complexe (AnalizaDate, OrdLista).

```
procedure CitNume (var Nx:Sir);
   var i:1..25;
   begin
      for i:=1 to 25 do
         if eoln then Nx[i]:=' '
         else read (Nx[i]);
      readln
   end;
procedure CitMult (n:integer; var m:Mulţime);
   var e:Numere;
      i:integer;
   begin
      m:=[];
      for i:=1 to n do
         begin
            repeat
               write ('Numărul', i:2, 'este:');
               readln(e);
            until ((e >=1) and (e <=Maxim) and (not (e in m))) ;
            m:=m+[e]
         end;
      writeln
   end;
procedure ScrieMult (m:Mulţime);
   var MTemp:Mulţime;
      i:integer;
   begin
      MTemp:=m;
      i:=1;
      while MTemp <> [] do
         begin
            if i in MTemp then
               begin
                  write(i,' ') ;
                  MTemp:=MTemp—[i]
               end;
            i:=i+1
```

```
      end
   end ;
function Cardinal (m:Mulţime): integer;
   var MTemp:Mulţime;
      n, i:integer;
   begin
      n:=0;
      MTemp:=m;
      i:=1;
      while MTemp <> [] do
         begin
            if i in MTemp then
               begin
                  n:=n+1;
                  MTemp:=MTemp-[i];
               end;
            i:=i+1;
         end;
      Cardinal:=n
   end;
```

15.3. PROGRAMUL COMPLET

Pentru exemplul luat în considerare forma completă a programului este următoarea :

```
program PronoExpres;
   const
      Maxim     = 45; {cel mai mare număr din set}
      NrMaxJuc  = 200; {numărul maxim de jucători}
      NrExtrase = 12;
      NrPropuse =  5;
   type
      Numere    = 1..Maxim;
      Mulţime   =set of Numere;
      Sir       =packed array [1..25] of char;
      Concurent =record
                    Nume: Sir;
                    MNrJuc, MNr0K: Mulţime;
                    nng: 0..NrPropuse;
                 end
      Registru= array [1..NrMaxJuc] of Concurent;
   var
      Lista: Registru;
      Urna: Mulţime;
      NrJuc: 1..NrMaxJuc;
function Cardinal (m:Mulţime):integer;
   var MTemp:Mulţime;
      n, i:integer;
   begin
      n:=0;
      MTemp:=m;
      i:=1;
```

```
         while MTemp <> [] do
            begin
               if i in MTemp then
                  begin
                     n:=n+1;
                     MTemp:=MTemp—[i];
                  end;
               i:=i+1;
            end;
         Cardinal:=n
      end;
   procedure CitNume (var Nx: Sir);
      var i:1 . .25;
      begin
         for i:=1 to 25 do
            if eoln then Nx[i]:=' '
            else read (Nx[i]);
         readln
      end;
   procedure CitMult (n:integer; var m:Mulţime);
      var e:Numere;
          i:integer;
      begin
         m:=[];
         for i:=1 to n do
            begin
               repeat
                  write('Numărul', i:2, 'este:');
                  readln(e);
               until ((e >=1) and (e < =Maxim) and (not (e in m)));
               m:=m+[e]
            end;
         writeln
      end;
   procedure ScrieMult, (m:Mulţime);
      var MTemp:Mulţime;
          i:integer;
      begin
         MTemp:=m;
         i:=1;
         while MTemp <> [] do
         if i in MTemp then
            begin
               write(i, ' ');
               MTemp:=MTemp—[i]
            end;
         i:=i+1
         end
      end;
   procedure Antet;
      var k:integer;
      begin
         for k:=1 to 5 do writeln;
         writeln ('************** PRONOEXPRES **************');
```

```
    for k:=1 to 5 do writeln
  end;
  procedure CitDate (var 1: Registru);
  var i:integer;
  begin
    writeln;
    writeln (' *urmează introducerea datelor jucătorilor*');
    writeln;
    write ('Număr de jucători ?:');
    readln(NrJuc) ; {s-a citit numărul de jucători}
    for i:=1 to NrJuc do
      with l[i] do
        begin
          writeln (' ————————————————————————————');
          write ('Numele jucătorului', i:4, '?');
          CitNume(Nume);
          writeln ;
          write ('Numerele propuse:');
          writeln;
          CitMult(NrPropuse, MNrJuc)
        end;
      {In cadrul instrucţiunii for au fost citite datele asociate tuturor jucăto-
      rilor}
    writeln;
    writeln ('Precizaţi numerele extrase!');
    writeln ;
    CitMult (NrExtrase, Urna);
    {au fost citite toate numerele extrase}
    writeln
  end;
procedure AnalizaDate (var 1:Registru):
  var i:integer;
  begin
    for i:=1 to NrJuc do
      with l[i] do
        begin
          MNr0K:=MNrJuc*Urna;
          nng:=Cardinla (MNrOK)
        end
  end;
procedure OrdLista (l:registru);
  var
    i:integer;
    Temp:Concurent;
    flag:boolean;
  begin
    repeat
      flag:=false;
      for i:=1 to NrJuc—1 do
        if Lista[i].nng <Lista[i+1] .nng then
          begin
            Temp:=Lista[i];
            Lista[i]:=Lista[i+1];
```

```
                Lista[i+1]:=Temp;
                flag:=true
              end
      until not flag;
   end;
procedure TipRez (l:Registru);
   var i:integer;
   begin
      writeln;
      writeln ('Urmează listarea rezultatelor');
      writeln;
      writeln ('Nr. crt.    Nume jucător     Nr. numere ',
      'Numerele');
      writeln ('                                ghicite ',
      'ghicite');
      writeln;
      for i:=1 to NrJuc do
         with l[i] do
            begin
               write (i:4,'     ', Nume, nng:5,'      ');
               ScrieMult (MNrOK);
               readln
            end;
      writeln;
      writeln ('*** SFÂRŞIT ***');
      readln
   end;
   {De aici începe programul principal}
   begin
      Antet;
      CitDate(Lista);
      AnalizaDate(Lista);
      OrdLista(Lista);
      TipRez(Lista)
   end.
```

Observaţii

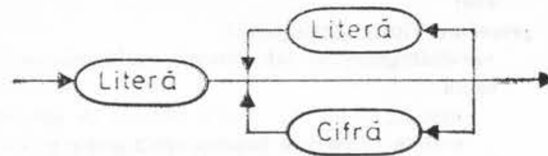
1. Pentru problema de faţă, aşa cum este ea formulată, ultimele patru proceduri (considerate în ordinea apelării din programul principal) nu aveau nevoie de parametru formal. Lista fiind practic unică şi deci putând fi declarată ca variabilă globală. În acest caz, grupul de instrucţiuni ce formează programul principal poate avea forma:

```
begin
   Antet;
   CitDate;
   AnalizaDate;
   OrdLista;
   Tiprez
end.
```

iar procedurile respective trebuie declarate **fără parametru formal**.

2. Programul prezentat este, desigur, perfectibil. Cititorul interesat poate relua o parte dintre etapele descrise anterior şi poate concepe, destul de uşor, o variantă, îmbunătăţită a acestui program.

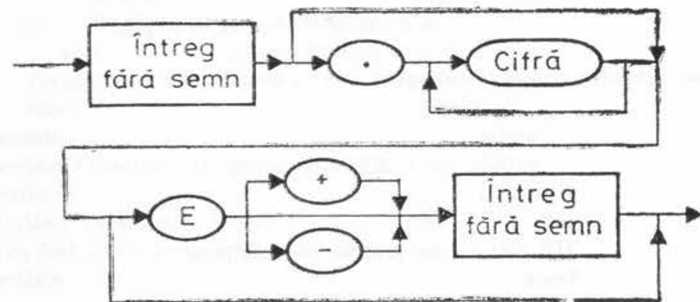
IDENTIFICATOR



ÎNTREG FĂRĂ SEMN



NUMĂR FĂRĂ SEMN



CONSTANTĂ FĂRĂ SEMN

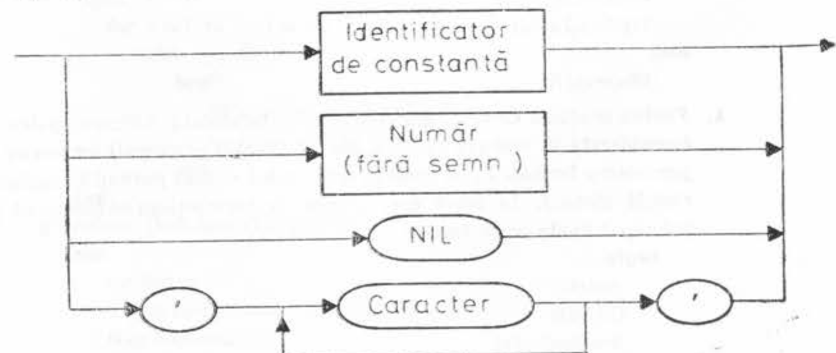
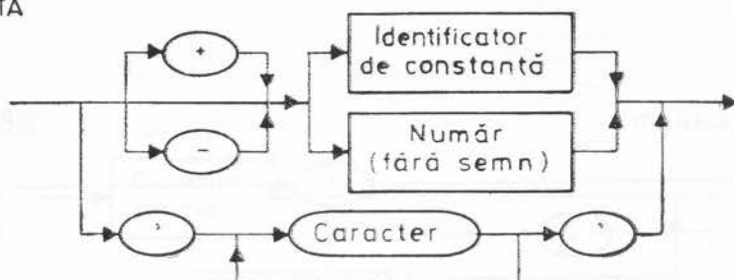
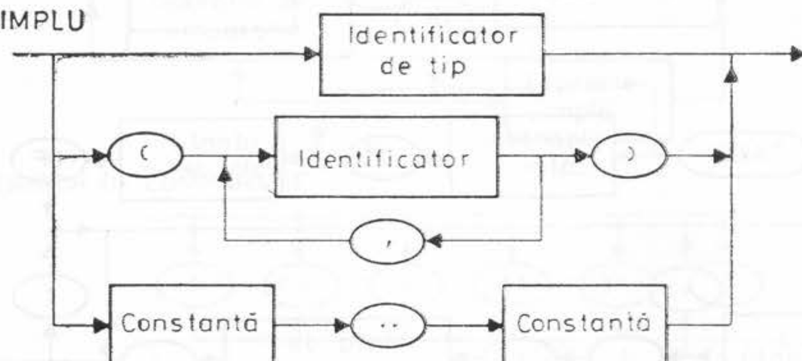


Fig. A.1.

CONSTANTĂ



TIP SIMPLU



TIP

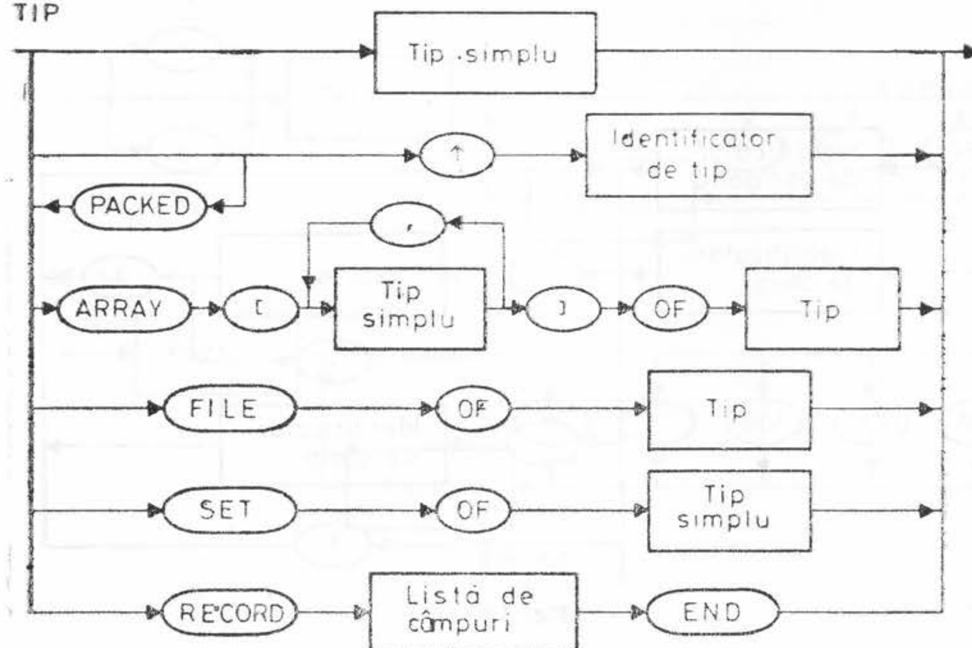
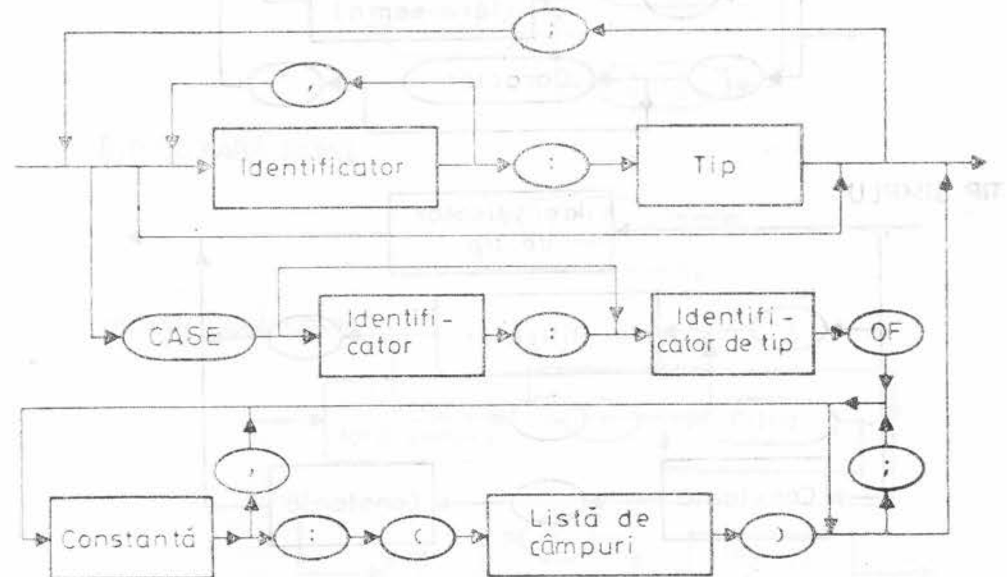


Fig. A.2.

LISTA DE CÂMPURI



VARIABILĂ

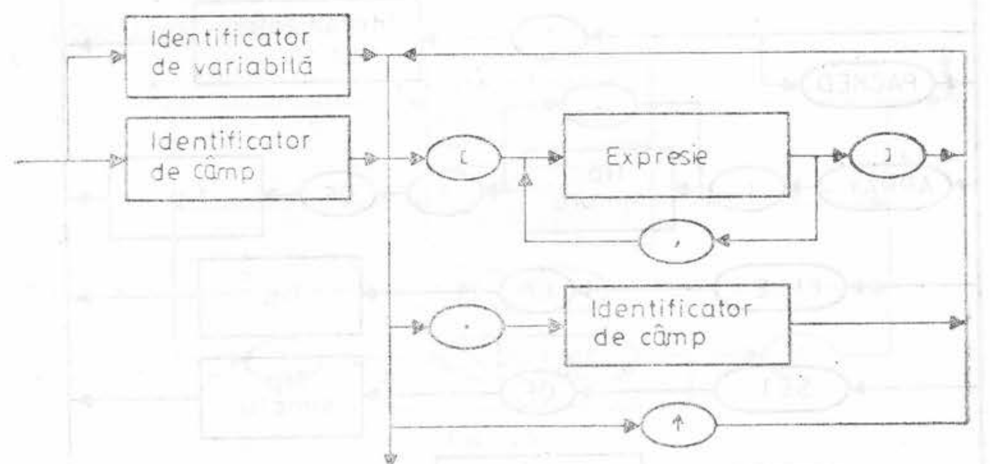
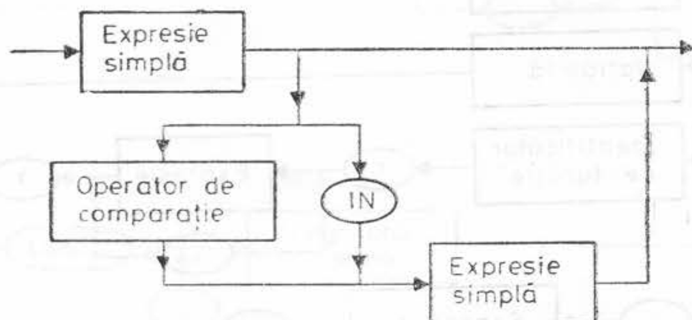
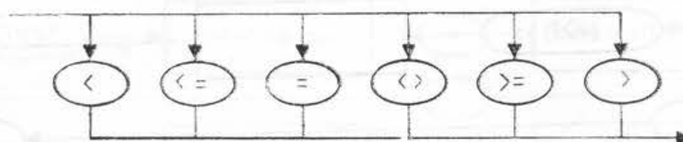


Fig. A.3.

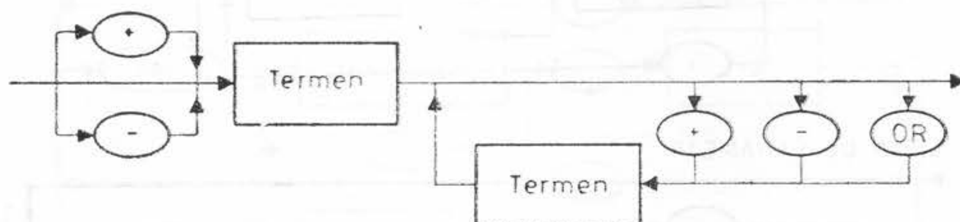
EXPRESIE



OPERATOR DE COMPARAŢIE



EXPRESIE SIMPLĂ



TERMEN

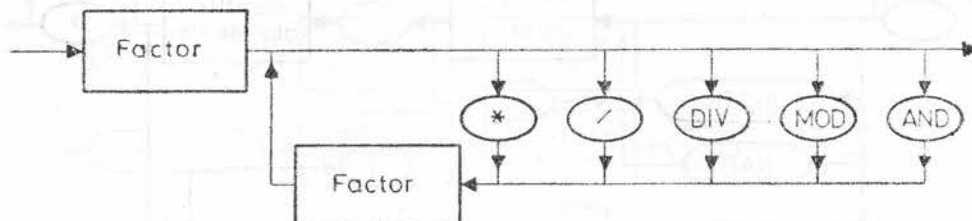


Fig. A.4.

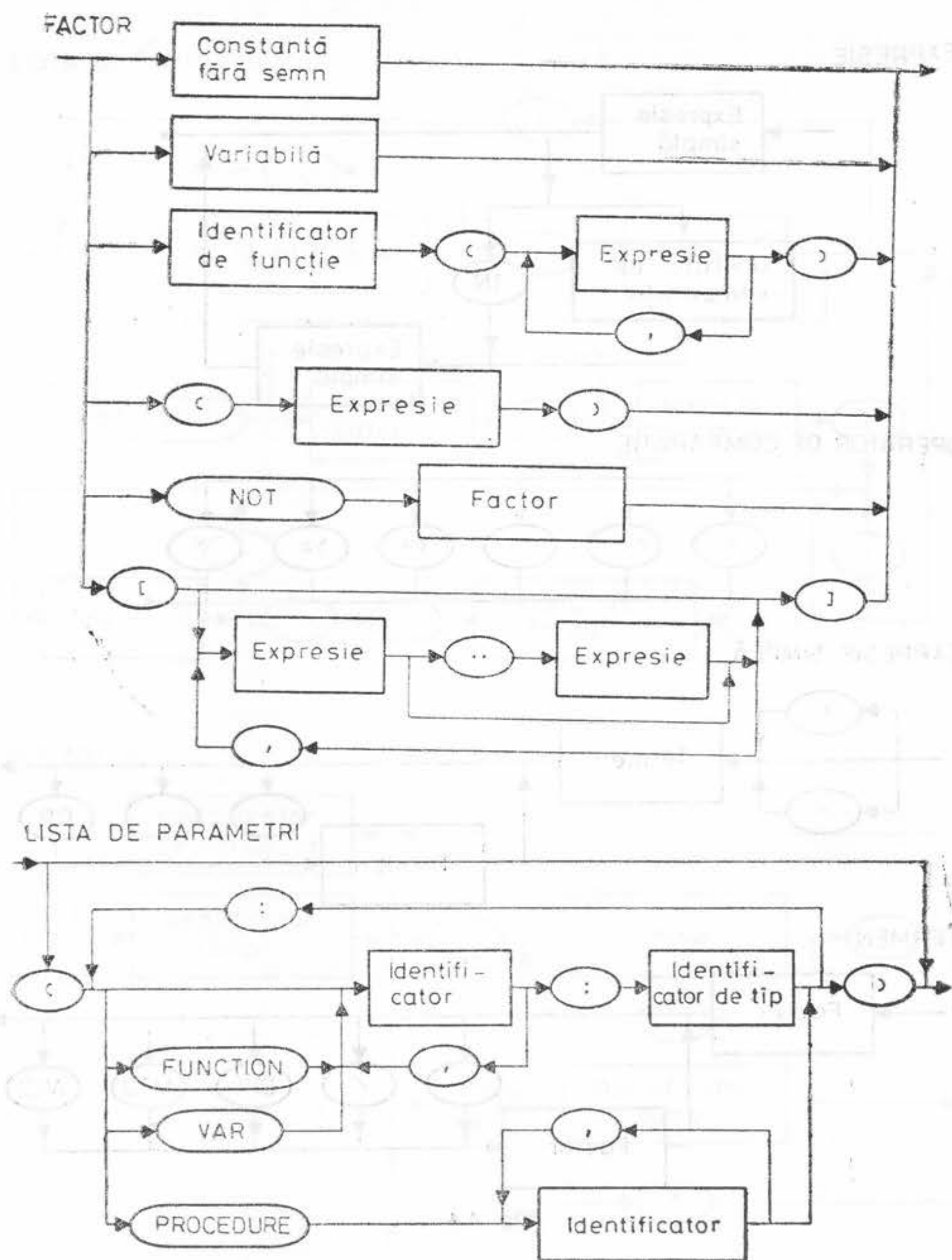
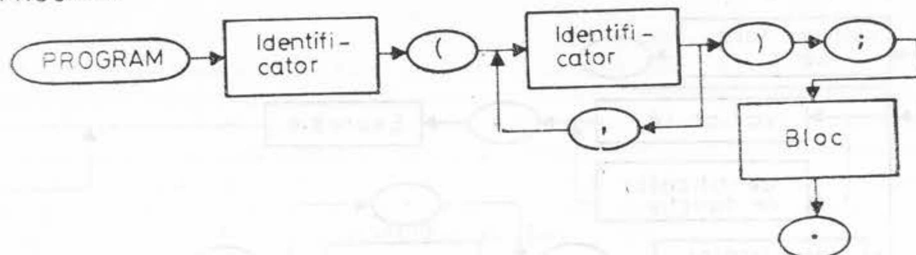

FACTOR
Constantă fără semn
Variabilă
Identificator de funcție
(
Expresie
,
)
(
Expresie
)
NOT
Factor
[
Expresie
..
Expresie
,
]
LISTA DE PARAMETRI
(
:
Identificator
:
Identificator de tip
,
FUNCTION
VAR
PROCEDURE
,
Identificator
)

Fig. A.5.

PROGRAM



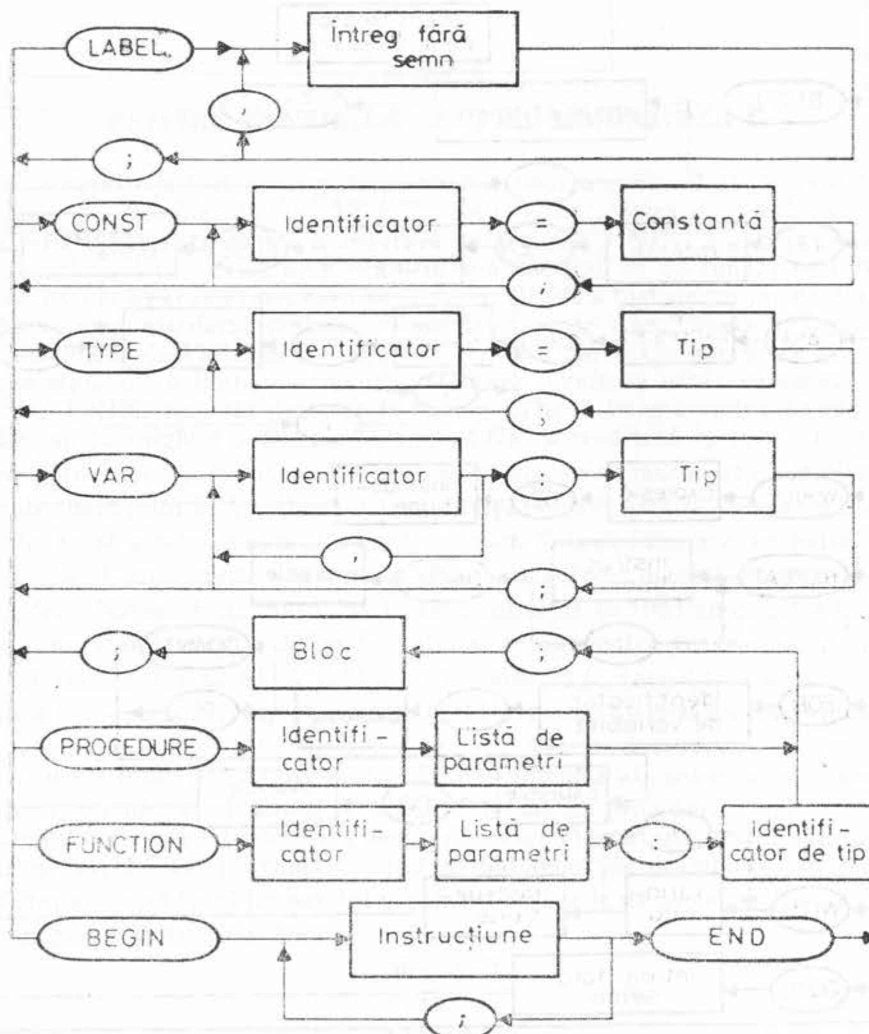


Fig. A.6.

INSTRUCȚIUNE

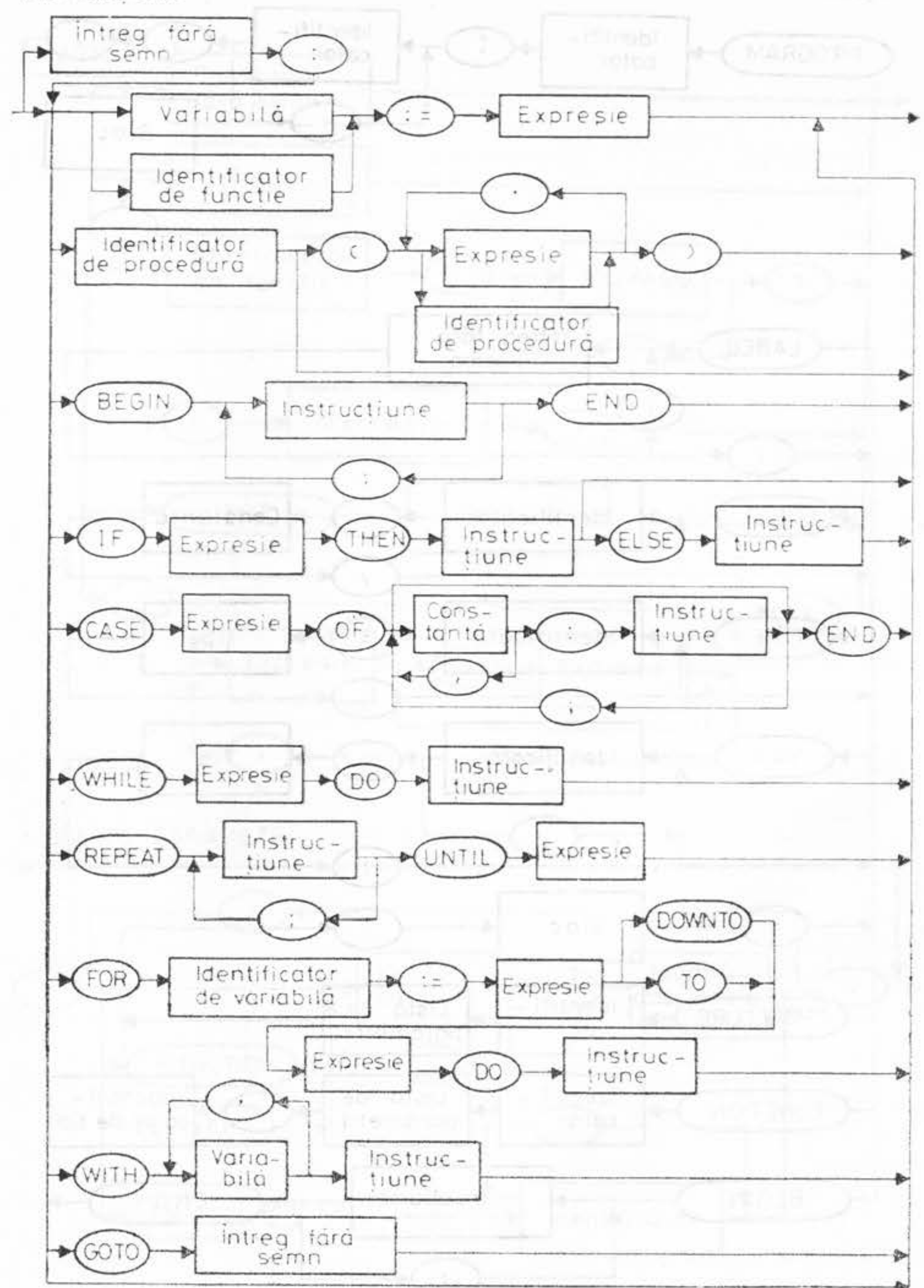


Fig. A.7.

PARTEA A II-A

LIMBAJUL DE PROGRAMARE C

CAPITOLUL 16

PRIVIRE GENERALĂ ASUPRA LIMBAJULUI C

Limbajul C a fost creat şi implementat pentru prima dată în anul 1972 de Dennis Ritchie de la AT & T Bell Laboratories pe echipamente DEC—PDP-11, care utilizau sistemul de operare UNIX. Limbajul C este rezultatul unui proces de dezvoltare care a început cu un limbaj mai vechi numit BCPL, apărut prima oară în Europa. BCPL a fost dezvoltat de Martin Richards şi a influenţat limbajul numit B creat de Ken Thomson. Limbajul B a condus la dezvoltarea limbajului C în anul 1970. Un număr de ani versiunea standard a limbajului C era versiunea furnizată odată cu sistemul de operare UNIX. Ea este descrisă în cartea „The C Programming Language" de Berian Kernighan şi Dennis Ritchie (1978) şi reeditată în 1988. O dată cu dezvoltatea microcalculatoarelor au fost create un număr mare de implementări ale limbajului C. Un element foarte important constă în compatibilitatea codului sursă a celor mai multe implementări. Totuşi, deoarece nu exista nici un standard, erau multe discrepanţe. Pentru a corecta această situaţie, ANSI (American National Standard Institute) a înfiinţat în 1983 un comitet pentru a crea un standard şi a defini limbajul C odată pentru totdeauna. Astfel a apărut standardul ANSI în 1988 pentru limbajul C, care diferă de versiunea iniţială în anumite privinţe. Care sunt caracteristicile de bază ale limbajului C ?

Limbajul C este un *limbaj de nivel mediu*. Asta nu înseamnă că el este mai „slab" decît limbajele de nivel înalt BASIC sau Pascal, sau că el este similar cu limbajele de asamblare care ridică probleme de programare utilizatorilor. Limbajul C se numeşte de nivel mediu deoarece el combină avantajele limbajelor de nivel înalt cu funcţionalitatea limbajelor de asamblare. În tabelul următor se arată locul limbajului C printre limbajele de programare.

- *Limbaje de nivel înalt* : Ada
  Modula-2
  Pascal
  COBOL
  FORTRAN
  BASIC

• *Limbaje de nivel mediu :* C
FORTH
Macro-asembler

• *Limbaje de nivel inferior :* Limbaje de asamblare.

Ca limbaj de nivel mediu, C permite manipularea biţilor, octeţilor şi adreselor — elemente de bază cu care operează calculatorul. Codul C este foarte portabil. Portabilitatea constă în posibilitatea de adaptare a software-ului pe diferite tipuri de calculatoare. Ca toate limbajele de nivel înalt suportă conceptul de *tipuri de date* (întreg, caracter, real). Limbajul C are posibilitatea să lucreze cu tipuri de date diverse (char, int, float, double) şi permite orice fel de conversii (într-o expresie pot apărea tipuri de date diferite). În versiunea originală de C, nu se făcea un test de compatibilitate între parametrii formali ai unei funcţii şi argumentele reale utilizate la apelul funcţiei. Astfel, de exemplu, se putea apela o funcţie utilizând un pointer ca parametru formal, fără a genera eroare dacă argumentul actual al funcţiei era un real. Eroarea apărea doar în faza execuţiei, creînd mari probleme la depanarea programelor. De aceea standardul ANSI a introdus conceptul de *prototip funcţie* care permite să fie raportate asemenea erori potenţiale încă din faza de compilare.

Limbajul C are doar 32 de cuvinte cheie. Spre comparare, limbajul BASIC pentru IBM-PC conţine în jur de 159 de cuvinte cheie.

Limbajul C este un *limbaj structurat*, ca şi alte limbaje (ALGOL, Pascal, Modula-2). El însă nu permite crearea şi declararea unor funcţii în interiorul altor funcţii. Trăsătura caracteristică a limbajelor structurate este compartimentarea codului şi datelor. Aceasta este capacitatea limbajului de a extrage şi de a ascunde de restul programului instrucţiunile şi informaţiile necesare pentru a realiza o sarcină specifică.

Un mod de a realiza această compartimentare constă în folosirea subrutinelor (funcţiilor) care lucrează cu variabile locale (temporare). Folosind variabilele locale se pot scrie subrutine astfel încât evenimentele care se petrec în interiorul lor nu dau naştere la efecte secundare în alte părţi din program. Această capacitate uşurează programelor C partajarea secţiunilor de cod. Astfel, dezvoltând funcţii separate trebuie doar să ştii ce face funcţia, nu cum face funcţia. Să precizăm că folosirea excesivă a variabilelor globale (variabile recunoscute în tot programul) poate cauza efecte secundare nedorite (oricine programează în BASIC este conştient de această problemă).

Un limbaj structurat permite o mare varietate de posibilităţi de programare prin construcţiile pe care le suportă. În cadrul limbajului structurat instrucţiunea **goto** este prohibită. Să dăm exemple de limbaje structurate şi nestructurate:

— *nestructurate*: FORTRAN, COBOL, BASIC

— *structurate:* Pascal, Ada, C, Modula-2

Componenta structurală principală a limbajului C este funcţia-subrutina independentă a limbajului C. În limbajul C, funcţiile sunt *blocuri* în care se desfăşoară toată activitatea programului. Ele permit definirea şi programarea separată a sarcinilor distincte ale unui program, permiţându-i acestuia să fie modular.

O funcţie odată creată, se va executa corect în diferite situaţii fără a da naştere la efecte secundare în alte părţi ale programului. Capacitatea de a crea funcţii independente este importantă mai ales în programele mari, unde altfel pot apărea influenţe accidentale între diferite părţi ale programului.

Un alt mod de structurare şi compartimentare a programelor în limbajul C constă în folosirea blocurilor de program (code block). Un astfel de bloc este format dintr-un grup de instrucţiuni înlănţuite logic şi tratate ca o unitate. În limbajul C un astfel de bloc este constituit dintr-un şir de instrucţiuni incluse între acolade ({bloc}). Cu ajutorul unor astfel de blocuri implementarea multor algoritmi se poate face cu claritate, eleganţă şi eficienţă.

Un factor determinant în răspândirea limbajului C în rândul programatorilor consta în posibilitatea utilizării lui cu succes în locul limbajului de asamblare, care este greu de folosit în aplicaţii (conţine un număr mare de instrucţiuni elementare şi nu este structurat). În plus, programele elaborate în limbaj de asamblare nu sînt portabile.

Limbajul C este singurul limbaj de programare structurat care permite un control riguros al hardware-ului şi al perifericelor (facilitate oferită de limbajele de asamblare). Limbajul C a fost folosit iniţial pentru scrierea programelor de sistem (sisteme de operare, editoare, compilatoare, etc.), dar odată cu creşterea popularităţii lui, programatorii au început să-l folosească şi la scrierea programelor de aplicaţii datorită în special eficienţei şi portabilităţii crescute a unor astfel de programe.

În **concluzie**, limbajul C este un *limbaj agreat de programatori*, permiţându-le acestora să alcătuiască cu uşurinţă în mod sistematic programe complexe. Fiecare programator îşi poate crea o bibliotecă de funcţii pe măsura personalităţii fiecăruia, care pot fi folosite în diferite situaţii.

Datorită acestor calităţi incontestabile *limbajul C a devenit limbajul de bază şi pentru programarea aplicaţiilor de timp real.*

CAPITOLUL 17

STRUCTURA PROGRAMELOR ÎN LIMBAJUL C. TIPURI DE DATE, OPERATORI ŞI EXPRESII

17.1. STRUCTURA PROGRAMELOR ÎN LIMBAJUL C

La fel ca în orice activitate de programare, prima acţiune pe care trebuie să o întreprindă programatorul este să scrie cu ajutorul unui editor programul, în conformitate cu regulile sintactice şi semantice aferente acestui limbaj. Se elaborează astfel aşa-numitul *„program sursă"*, care pentru a fi executat trebuie să parcurgă o serie de etape:

— *etapa de* **compilare** care are două componente:

* rezolvarea directivelor către preprocesor prin expandarea în codul-sursă a unor forme reduse determinate de aceste directive (dacă există);
* transpunerea programului sursă în **„program obiect"**;

— *etapa de* **link-editare** care presupune legarea programului obiect obţinut mai sus cu bibliotecile de sistem şi transformarea lui într-un **„program executabil"**;

— *etapa de* **lansare în execuţie.**

Modul în care se execută aceste acţiuni este specific tipului de calculator şi de sistem de operare care conţine limbajul de programare. În cele ce urmează nu vom insista asupra acestor elemente, noi având în vedere problemele generale de programare pentru elaborarea programului sursă, probleme valabile, indiferent de maşina pe care se lucrează.

Să scriem şi să comentăm un program foarte simplu scris în C, pentru a desprinde o serie de reguli generale referitoare la structura acestor programe.

Exemplu 17.1

```
main()
{
      printf ("Noi învăţăm limbajul C!");
}
```

Presupunând parcurse în mod corect etapele de compilare și link-editare a programului, la lansarea lui în execuție va apare mesajul:

Noi învățăm limbajul C!

O primă observație importantă pe care trebuie să o facem este aceea că orice program în C este compus dintr-o serie de entități numite „**funcții**". O funcție, pentru a o deosebi de un cuvânt cheie oarecare al limbajului, se notează cu numele funcției, urmat de două paranteze rotunde:

nume funcție();

● **Funcția main()** este aceea către care sistemul de operare transferă controlul atunci când se lansează în execuție programul. În anumite situații, în interiorul parantezelor aceste funcții pot exista și parametri. Începutul și sfârșitul corpului funcției formează un bloc marcat de paranteze acolade. Aceste paranteze au un rol similar instrucțiunilor BEGIN și END din Pascal. Așa cum vom vedea în continuare, parantezele acolade se utilizează și pentru delimitarea altor blocuri întîlnite de exemplu în structurile interative sau decizionale.

Linia de program:

printf („Noi învațam limbajul C!");

(corpul funcției **main()**) formează o instrucțiune și ea este obligatoriu terminată cu semnul (;). Trebuie făcută observația că spațiile albe (blank, tab, newline) sunt invizibile pentru compilator. Astfel, programul mai putea fi scris astfel:

main(){printf („Noi învățăm limbajul C!");}

Prin delimitarea blocurilor cu ajutorul parantezelor acolade programul devine mai ușor de citit și de interpretat. O sinteză a celor spuse mai sus este prezentată în figura 17.1.

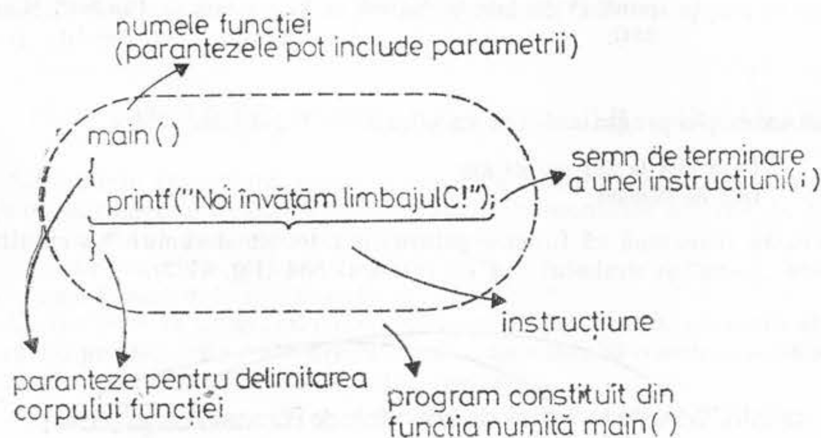


Fig. 17.1.

În programul de mai sus, „**printf()**" este numele unei funcții, la fel ca **main()**, dar care are argumente între paranteze și anume un șir de caractere. Aceasta este o funcție sistem (nu este creată de programator) foarte versatilă, care permite afișarea la consolă (display) a constantelor, variabilelor și caracterelor.

Să explorăm posibilitățile acestei funcții.
Considerăm următorul program:

Exemplul 17.2

```
main()
{
        printf ("Acesta este numărul cinci: %d",5);
}
```

În urma lansării în execuție a programului va apare la consolă:

```
Acesta este numarul cinci: 5
```

Funcția printf() are în acest caz două argumente separate prin virgulă:

- șirul de caractere "Acesta este numărul cinci: %d";
- valoarea întreagă 5.

Simbolul „%d" permite scrierea parametrului din dreapta virgulei — valoarea 5 — alături de șirul de caractere. Acesta reprezintă unul din „formatele specifice" utilizate de funcția printf() și anume acela pentru scrierea valorilor întregi.

Pentru a înțelege și mai bine modul în care lucrează funcția printf() să mai luăm un exemplu.

Exemplul 17.3

```
main()
{
        printf (" %s este la %d mii de km\n față de Pământ", "Luna",
        384);
}
```

În urma execuției programului se va afișa:

```
Luna este la 384 mii de km
față de Pământ
```

Această înseamnă că funcția **printf()** a înlocuit simbolul %s cu șirul de caractere „Luna" și simbolul %d cu întregul 384 (fig. 17.2).

printf("%s este la %d mii de km\n fațăde Pământ","Luna",384);

Fig. 17.2.

Simbolul '/n' reprezintă caracterul **„newline"** și are efect de **carriage return and linefeed** (rând nou și de la capăt). Acesta este motivul că textul a fost afișat pe două rânduri, deși el a fost scris în program într-un singur rând. El formează o așa-numită **secvență „escape"**.

Să considerăm programul următor:

Exemplul 17.4.

```
main()
{
        printf("Litera %c se", 'f');
        printf ("pronunță %s", "ef");
}
```

În urma execuției va apărea:

```
Litera f se pronunță ef
```

adică scrierea se face pe un singur rând deși în program sunt două linii de instrucțiuni. Aici 'f' este caracter și se scrie cu formatul %c, iar „ef" este șir de caractere și se va afișa cu ajutorul formatului %s. Se observă de asemenea modul de scriere în program a caracterelor (delimitate de ' ') și a șirurilor de caractere (delimitate de " ").

Observație: În cazul anumitor implementări ale limbajului C, există posibilitatea ca programele de mai sus (precum și mai multe dintre cele care vor urma) să genereze erori la faza de compilare și deci să nu poată fi executate.

Pentru a fi înlăturate erorile de compilare este necesar să mai fie introdusă înainte de main() o linie de program conținând:

#include <stdio.h>

în cazul utilizării funcțiilor printf()) și scanf() și încă o linie de program cu :

#include <conio.h>

în cazul utilizării funcției getche().

Motivul introducerii acestor linii de program va fi explicat în paragraful 19.12.

Înainte de a trece la probleme mai profunde, legate de programarea în limbajul C, este necesar să lămurim trei **aspecte importante ale acestui limbaj:** *tipurile de date* cu care operează limbajul, *modul de citire și afișare* al lor și *operatorii utilizați* în combinarea lor.

17.2. VARIABILE ; TIPURI DE VARIABILE ; DECLARARE

Variabilele reprezintă spații în memoria calculatorului având același rol în timp, dar care pot conține valori diferite la momente diferite de timp. În legătură cu ele se pun următoarele probleme:

— ce fel de variabile recunoaște limbajul?
— cum sunt memorate aceste date în calculator?

Să revenim la programul din exemplul 17.2. Cifra 5, preluată și afișată de funcția **printf()**, este o valoare întreagă constantă. Să rescriem acest program utilizând o variabilă în locul acestei constante.

Exemplul 17.5.

```
main()

{
        int num;
        num=5;
        printf ("Acesta este numărul cinci: %d", num);
}
```

În urma execuției, la consolă va apare același mesaj

Acesta este numarul **cinci**: 5

Programul conține însă câteva elemente noi. În primul rând instrucțiunea:

int num;

prin care este declarată drept întreg variabila *num*. Deci *num* este numele variabilei iar **int** este tipul ei (întreg).
În a doua instrucțiune:

num=5;

se atribuie valoarea 5 acestei variabile. Astfel operatorul (=) este *operatorul de atribuire* similar operatorului (:=) din Pascal.
În aceste condiții devine clar modul de lucru al funcției **printf()** din instrucțiunea următoare a programului; va afișa valoarea varibilei *num* care este 5.
Acțiunea de declarare a variabilelor este obligatorir în limbajul C. Declararea constă în precizarea tipului variabilei (int) și a numelui său (num). Mai multe variabile de același tip pot fi declarate în aceeași linie de program, de exemplu:

int num1, num2, num3;

Numele unei variabile poate fi format dintr-unul sau mai multe caractere alfanumerice, începând cu un caracter alfabetic, având cel mult 31 de caractere. Caracterul — (subliniere) este considerat literă.

Exemple:

Corect	*Incorect*
num1	1 num
cursor—dr	cursor. . dr
	cursor!dr

Declarația variabilei determină compilatorul să-i aloce un spațiu corespunzător de memorare.
Limbajul C recunoaște cinci **tipuri de variabile**:
— caracter: **char**
— întreg: **int**
— tip de variabilă neprecizat sau inexistent: **void**
— real în virgula mobilă în simplă precizie: **float**
— real în virgula mobilă în dublă precizie: **double**
Modul de memorare a acestor tipuri de date depinde de tipul calculatorului și de varianta de implementare a limbajului C. Modul de implementare al lor poate fi modificat prin utilizarea unor declarații suplimentare, cum ar fi:
— **signed** (cu semn)
— **unsigned** (fără semn)
— **long** (lung)
— **short** (scurt).
Apel nd la o implementare uzuală a limbajului C pe echipamente PC, compatibile IBM sub sistemul de operare MS-DOS, tipurile de date definite, de standardul ANSI și recunoscute de limbaj au reprezentarea din tabelul următor:

Tip	Reprezentare — în biți —	Rang
char	8	$-128 \div 127$
unsigned char	8	$0 \div 255$
signed char	8	$-128 \div 127$
int	16	$-32768 \div 32767$
unsigned int	16	$0 \div 65535$
signed int	16	$-32768 \div 32767$
short int	16	$-32768 \div 32767$
unsigned short int	16	$0 \div 65535$
signed short int	16	$-32768 \div 32767$
long int	32	$-2.147.483.648 \div 2.147.483.647$
signed long int	32	$-2.147.483.648 \div 2.147.483.647$
unsigned long int	32	$0 \div 4.294.967.295$
float	32	$10^{-37} \div 10^{37}$ (6 digiți precizie)
double	64	$10^{-308} \div 10^{308}$ (10 digiți precizie)
long double	80	$10^{-4932} \div 10^{4932}$ (15 digiți precizie)

Exemple de declarații de variabile:
int i, j, k;
unsigned int 1;
double val, set;
float time ;

Să dăm un exemplu de program care utilizează declarații multiple de variabile.

Exemplul 17.6.

```
main()
{
        int ev;
        char poz;
        float timp ;
        ev=5;
        poz='M';
        timp=12.30;
        printf("Evenimentul %c are numarul %d", poz, ev);
        printf("si a avut loc la %f", timp);
}
```

În urma execuției acestui program se va afișa:

Evenimentul M are numărul 5 și a avut loc la 12.300000

Formatul %f s-a folosit în exemplul 17.6 pentru afișarea unui număr real în virgulă mobilă (valoarea variabilei timp declarată **float.**)

Putem prezenta acum formatele specifice utilizate de funcția **printf()**:

%c — afișarea unui caracter unic
%s — afișarea unui șir de caractere
%d — afișarea unui număr întreg (în baza zece) cu semn
%i — afișarea unui număr întreg (în baza zece) cu semn
%u — afișarea unui număr întreg (în baza zece) fără semn
%f — afișarea unui număr real (notație zecimală)
%e — afișarea unui număr real (notație exponențială)
%g — afișarea unui număr real (cea mai scurtă reprezentare dintre %f și %e)
%x — afișarea unui număr hexazecimal întreg fără semn
%o — afișarea unui număr octal întreg fără semn
%p — afișarea unui pointer (a unei adrese)

În plus se pot folosi următoarele prefixe:

l cu d, i, u, x, o — urmează să se afișeze o dată de tip long
l cu f, e, g — urmează să se afișeze o dată de tip double
h cu d, i, u, x, o — urmează să se afișeze o dată de tip short
L cu f, e, g — urmează să se afișeze o dată de tip long double.

Exemple

%ld — se va afișa o dată de tip long int
%hu — se va afișa o dată de tip short unsigned int

În exemplul 17.6 se observă la afișarea valorii reale că apar șase zecimale în loc de două câte au fost atribuite inițial acestei variabile. Pentru suprimarea celor patru zerouri inutile, formatul de scriere %f nu trebuie lăsat liber (caz în care funcția printf() afișează mereu cu șase zecimale), ci trebuie însoțit de o notație suplimentară specificând **dimensiunea câmpului de afișare** a datelor și **precizia** de afișare a acestora. Aceasta se introduce între simbolul % și simbolul f și are forma generală %**-a.bf**, cu a și b numere întregi având semnificațiile arătate în figura 17.3.

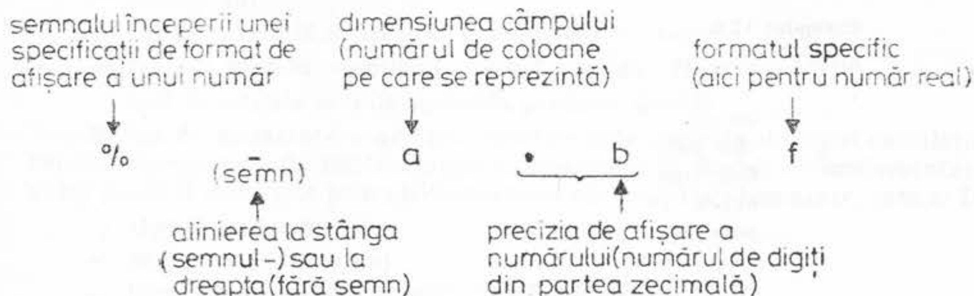


Fig. 17.3.

Astfel, în exemplul 17.6, dacă în locul specificației de format %f am fi trecut %5.2f, la consolă ar fi apărut mesajul:

Evenimentul M are numărul 5 și a avut loc la 12.30

Am specificat că numărul atribuit variabilei time se scrie aliniat la dreapta pe 5 coloane (el are 5 caractere) cu 2 zecimale. Tot același efect asupra scrierii l-ar fi avut și specificația %.2f, din care lipsește cifra reprezentând numărul de coloane. Desprindem astfel o regulă foarte importantă aplicată la scrierea numerelor reale cu ajutorul funcției printf().

Dacă numărul întreg **a** din specificația de format este mai mare decît numărul de caractere de scriere a valorii reale date, atunci scrierea se va face pe un număr de coloane egal cu **a**, cu aliniere la stânga sau la dreapta (după cum **a** este precedat de semnul — sau + (eventual fără semn)).

Dacă numărul **a** este mai mic decît numărul de caractere, atunci valoarea lui **a** se ignoră, scrierea făcându-se, fără aliniere, pe un număr de coloane egal cu numărul de caractere al valorii reale. Pentru înțelegerea mai bună a modului în care acționează această specificație de format să luăm un alt exemplu.

Exemplul 17.7.

```
main()
{
        float val;
        val=10.12304;
        printf ("%8.1f%8.1f\n“ 12.5,525.7);
        printf ("%—8.1f%—8.1f\n“ 12.5,525.7);
        printf ("%f\n“, val);
        printf ("%5.2f\n“, val);
        printf ("%10f\n“, val);
        printf ("%012f\n“, val) ;
        printf ("%—10f\n“, val);
        printf ("%—012f\n“, val);
}
```

În urma executării programului, vor rezulta următoarele modalități de afișare a numerelor menționate.

				1	2	·	5				5	2	5	·	7
1	2	·	5					5	2	5	·	7			
1	0	·	1	2	3	0	4	0							
1	0	·	1	2											
	1	0	·	1	2	3	0	4	0						
0	0	0	1	0	·	1	2	3	0	4	0				
1	0	·	1	2	3	0	4	0							
1	0	·	1	2	3	0	4	0	0	0	0				

Să mai observăm că punerea unui zero înaintea numărului care specifică dimensiunea câmpului de scriere determină la scriere umplerea cu zero a spațiilor goale.

- Specificarea dimensiunii câmpului de afișare a datelor precum și a preciziei de afișare se poate face și în cazul formatelor de întreg sau șir de caractere cu următoarele semnificații:

1. **Specificarea dimensiunii câmpului de afişare** (*întregul* **a** *din figura* 17.3):
— dacă numărul de caractere al şirului sau al întregului este mai mare decât **a**, şirul sau întregul se vor afişa în întregime fără a ţine cont de această specificare.
— dacă numărul de caractere al şirului sau al întregului este mai mic decât valoarea **a**, şirul sau întregul se vor afişa pe un număr de coloane egal cu **a** cu aliniere la dreapta dacă semnul este (+) sau lipseşte şi cu aliniere la stânga dacă semnul este (—), restul spaţiilor rămânând goale.

Exemplul 17.8.

```
main()
{
        int num;
        num=12345
        printf(" %d\n", num);
        printf (" %3d\n", num);
        printf (" %10d\n, num);
        printf(" %03d\n", num);
        printf(" %010d\n", num);
        printf (" %—10d\n", num);
```

Notă. În acest exemplu, precum şi în următoarele, liniile despărţitoare nu apar la afişare ; ele sunt trecute doar pentru a avea o imagine mai clară asupra modului de afişare a numerelor şi caracterelor.

În urma execuţiei programului vor rezulta următoarele modalităţi de scriere a valorii numerelor:

1	2	3	4	5					
1	2	3	4	5					
					1	2	3	4	5
1	2	3	4	5					
0	0	0	0	0	1	2	3	4	5
1	2	3	4	5					

Exemplul 17.9.

```
main()
{
        printf (" %s\n", "PROGRAM");
        printf (" %3s\n", "PROGRAM");
        printf (" %15s\n", "PROGRAM");
        printf (" %—15s\n", "PROGRAM");
        printf (" %015s\n", "PROGRAM");
        printf (" %03s\n", "PROGRAM");
}
```

La execuție se va afișa:

P	R	O	G	R	A	M								
P	R	O	G	R	A	M								
								P	R	O	G	R	A	M
P	R	O	G	R	A	M								
								P	R	O	G	R	A	M
P	R	O	G	R	A	M								

2. **Specificarea preciziei** (*întregul* **b** *din figura 17.3*):
— în cazul unui șir de caractere precizia determină lungimea maximă a câmpului de reprezentare.
ex: %5.7s determină afișarea unui șir de minimum 5 caractere dar nu mai mare de 7 caractere cu aliniere la dreapta.
— în cazul unui întreg, precizia determină numărul minim de digiți de afișare a numărului; dacă numărul nu are atâția digiți se adaugă zerouri.

Exemplul 17.10.

```
main()
{
        printf ("%.4f\n", 12.1234567);
        printf ("%3.8d\n", 1000);
        printf ("10.8d\n", 1000)
        printf ("%10.15s\n", "Acesta este un text");
        printf ("10.15s\n", "PROGRAM");
        printf ("%3.1d\n", 1000);
}
```

În urma execuției programului se vor afișa:

1	2	3	.	1	2	3	5							
0	0	0	0	1	0	0	0							
		0	0	0	0	1	0	0	0					
A	c	e	s	t	a		e	s	t	e		u	n	
			P	R	O	G	R	A	M					
1	0	0	0											

17.4. SECVENȚE escape

Să revenim asupra unui alt element apărut în exemplul 17.3. Am văzut cum caracterul '\n' inserat în șirul de caractere constituind parametrii funcției **printf()** a determinat combinația *carriage return — linefeed*. Acest caracter este un exemplu de secvență **escape**, numită așa deoarece simbolul

(\\) — **backslash** — este considerat caracter escape, care determină o abatere de la interpretarea normală a șirului. În cele ce urmează sunt prezentate și alte secvențe escape:

\n newline (determină carriage return-linefeed)
\t tab (determină saltul cursorului din 8 în 8 coloane începând cu capătul din stânga al consolei)
\b backspace (determină revenirea cursorului o coloană spre stânga)
\f formfeed (avansează la capătul paginii următoare la imprimantă)
\' apostrof simplu
\" (ghilimele)
\\ backslash
\xdd cod ASCII în notație hexazecimala (fiecare d reprezintă un digit); permite afișarea la consolă a caracterelor grafice utilizând codul hexazecimal
\ddd cod ASCII în notație octală
\r carriage return
\a alarmă sonoră (bell)

17.5. FUNCȚIA scanf()

O altă funcție foarte importantă a limbajului C este funcția de introducere a datelor **scanf()**. Limbajul C conține o clasă mare de funcții de intrare și de ieșire, dar **printf()** și **scanf()** sunt *cele mai versatile și pot manipula toate tipurile de variabile*. Să vedem un program care utilizează funcția **scanf()**.

Exemplul 17.11.

```
main()
{
    float ani, zile;
    printf ("Scrieți vârsta în ani:");
    scanf ("%f", &ani);
    zile=ani*365;
    printf ("Aveti virsta de %.1f zile:", zile);
}
```

În urma interacțiunii programului cu operatorul ar putea rezulta următoarea execuție a programului:

```
Scrieți virsta in ani: 42.5
Aveti virsta de 15512.5 zile
```

Cifra 42.5 este introdusă de operator de la tastatură după apariția la consolă a mesajului

```
Scrieți vârsta în ani:
```

În acest program, pe lângă funcția **scanf()**, mai apar două alte elemente noi:

— *operatorul de multiplicare* (*)
— *operatorul adresă* (&).

Să reținem pentru moment faptul foarte important că funcția **scanf()** cere utilizarea simbolului (&) înaintea numelui fiecărei variabile.

Astfel, argumentele funcției **scanf()** sunt: un șir de caractere care conține specificarea de format % și adresa variabilei *ani*. Specificarea formatelor pentru funcția **scanf()** este similară, cu mici excepții, cu cea pentru funcția **printf()** (de exemplu **scanf()** nu acceptă formate %i sau %g).

Funcția **scanf()** poate accepta în același timp mai multe intrări de date de tipuri diferite.
Să rescriem programul din exemplul 17.6.

Exemplul 17.12

```
main()
{
    int ev;
    char poz;
    float timp;
    printf ("Introduceti poz, ev, timp:");
    scanf ("%c %d %f", &poz, &ev., &timp);
    printf ("Evenimentul %c are numărul %d", poz, ev);
    printf ("și a avut loc la %f", timp);
}
```

Execuția programului poate fi:

```
Introduceți poz, ev, timp: M 5 12.30
Evenimentul M are numărul 5 și a avut loc la 12.30
```

Variabilele *poz. ev, timp,* au fost introduse de la consolă. Separarea lor corectă la introducere se poate face numai prin **spațiu** (blank), **return** sau **tab,** orice alt separator (linie, virgula) nerealizând această separare. Se pot introduce datele și separate prin (:) dacă în funcția **scanf()** se specifică aceasta în mod expres (între simbolurile de formate se va pune (:)). În figura 17.4 se arată cum lucrează funcția **scanf()**.

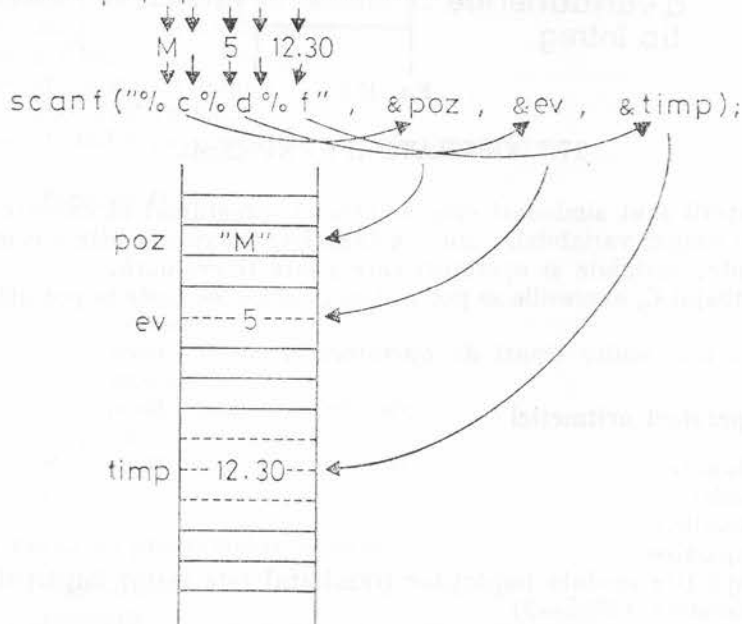


Fig. 17.4.

17.6. OPERATORUL DE ADRESARE (&)

Din motive care vor fi precizate în capitolul relativ la pointeri, compilatorul de C cere ca argument al funcției **scanf()** adresa variabilei și nu numele ei. Pentru aceasta funcția **scanf()** utilizează „ampersandul" (&), care precede numele variabilelor utilizate ca argumente — fig. 17.4. În exemplul următor se va vedea mai bine rolul acestui operator:

Exemplul 17.13

```
main()
{
        int num;
        num=2;
        printf "Valoare= %d; adresa= %x", num, &num);
}
```

În urma execuției poate apare:

```
Valoarea=2; adresa=1360
```

Se observă astfel că „num" este numele variabilei și are în acest caz valoarea 2, iar „&*num*" este adresa din memorie (1360), la care este depusă această valoare — figura 17.5. Valoarea adresei a fost afișată în cod hexazecimal.

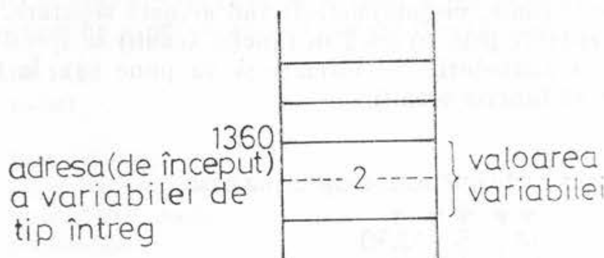


Fig. 17.5.

17.7. OPERATORI ; EXPRESII

Operatorii sunt simboluri care determină programul să execute o serie de operații asupra variabilelor sau constantelor. O expresie este o combinație de constante, variabile și operatori care poate fi evaluată.

În limbajul C, expresiile se pot utiliza în orice loc unde se pot utiliza variabile.

Există mai multe tipuri de operatori.

17.7.1. Operatori aritmetici

\+ adunare
— scădere
* înmulțire
/ împărțire
% împărțire modulo împărțitor (rezultatul este restul împărțirii — de exemplu 13%5=3)

Nu există operator pentru ridicare la putere. Ordinea operațiilor este cea binecunoscută, iar pentru modificarea acestei ordini se pot folosi paranteze.

Exemplul 17.14. *Transformarea unei temperaturi din grade Fahrenheit în grade Celsius.*

```
main()
{
    int ftemp, ctemp;
    printf ("Introduceţi temperatura în grade Fahrenheit:");
    scanf ("%d", &ftemp);
    ctemp=(ftemp—32)*5/9;
    printf ("Temperatura în grade Celsius este %d", ctemp);
}
```

În urma execuţiei se poate obţine:

```
Introduceţi temperatura în grade Fahrenheit: 80
Temperatura în grade Celsius este 27
```

17.7.2. Operatori de atribuire aritmetică

În orice limbaj de programare de niel înalt are sens următoarea instrucţiune:

val=val+num;

prin care la valoarea variabilei *val* se adaugă valoarea variabilei „*num*" şi rezultatul „se depune în memorie la adresa la care este memorată variabila *val*.

În limbajul C, această instrucţiune de atribuire se mai poate scrie prescurtat astfel:

val+=num;

unde combinaţia += poartă denumirea de *operator de atribuire cu adunare*.

Operatorii de atribuire în limbajul C sunt următorii:

+= atribuire cu adunare
—= atribuire cu scădere
*= atribuire cu înmulţire
/= atribuire cu împărţire
%= atribuire cu împărţire modulo împărţitor

Exemplul 17.15.

```
main()
{
    int val=0;
    int num=10;
    printf ("Total= %d\n", val);
    val+=num;
    printf ("Total= %d\n", val);
    val*=num;
    printf ("Total= %d\n", val);
}
```

În urma execuţiei programului rezultă:

```
Total=0
Total=10
Total=100
```

17.7.3. **Operatori de incrementare-decrementare**

În orice limbaj de nivel înalt putem scrie următoarea instrucțiune:

val=val+1;

În limbajul C, în locul acestei instrucțiuni, se poate folosi o formă prescurtată și anume:

val ++

Operatorul (++) poartă denumirea de **operator de incrementare.** Există de asemenea și un **operator de decrementare** (− −):

val − −;

care este echivalentul instrucțiunii

val=val−1.

Pentru a ne da seama mai bine cum lucrează acești operatori, să dăm un exemplu:

Exemplul 17.16.

```
main()
{
        int num=0;
        printf ("Număr= %d\n", num);
        printf ("Număr= %d\n", num++);
        printf ("Număr= %d\n", num);
}
```

Ce va afișa la execuție acest program?

```
Număr=0
Număr=0
Număr=1
```

Aceasta deoarece la prima cerere de afișare, variabila „num" are valoarea zero; la a doua cerere de afișare variabila „num" are la început tot valoarea zero pe care programul o afișează după care se face incrementarea variabilei care capătă valoarea 1. La a treia cerere de afișare va apare valoarea 1 a variabilei.

Dacă însă același program se va scrie ca în exemplul 17.17, execuția va fi diferită.

Exemplul 17.17.

```
main()
{
        int num=0;
        printf ("Număr= %d\n", num);
        printf ("Număr= %d\n", ++num);
        printf ("Număr= %d\n", num);
```

La execuție va apare:

```
Număr=0
Număr=1
Număr=1
```

Aceasta deoarece la a doua cerere de scriere a variabilei *num*, întâi se face incrementarea variabilei *num* (operatorul de incrementare ++ apare înaintea variabilei) și apoi se va scrie valoarea ei.

În mod similar lucrează şi operatorul de decrementare (— —).

Observaţie:

— unele implementări mai vechi ale limbajului C nu permit declararea şi iniţializarea unei variabile în aceeaşi linie de program ca în exemplele 17.15. 17.16 şi 17.17. Sunt necesare două instrucţiuni:

— una de declarare: int num;

— alta de iniţializare: num=0.

17.7.4. Operatori relaţionali

Aceştia permit stabilirea unei relaţii de ordine între variabile.

Exemplul 17.18.

```
main()
{
        int num=15;
        printf ("num este mai mic ca 21? %d\n" num<21);
        num=30;
        printf ("num este mai mic ca 21? %d\n", num<21);
}
```

La execuţie va apare:

```
num este mai mic ca 21? 1
num este mai mic ca 21? 0
```

În acest program funcţia **printf()** evaluează expresia "num<21" şi afişează valoarea ei logică:

1 dacă relaţia este adevărată

0 dacă relaţia este falsă

În limbajul C valorile logice „adevărat" şi „fals" se reprezintă prin valorile întregi 1, respectiv 0.

Simbolul (<) este un operator relaţional. **Operatorii relaţionali întâlniţi în limbajul C** sunt următorii:

< mai mic strict

> mai mare strict

<= mai mic sau egal

>= mai mare sau egal

== egal cu

!= diferit de

Se poate face remarca că valoarea 0 se generează ori de câte ori relaţia este falsă şi se generează valoarea 1 într-un singur caz, când relaţia este adevărată.

În ceea ce priveşte ordinea operaţiilor, operatorii aritmetici au prioritate faţă de operatorii relaţionali.

Exemplul 17.19.

```
main()
{
        printf ("Răspunsul este %d\n", 2+1<4); /* întîi se va calcula */
        ;                                       /*2+1=3, care se va */
        ;                                       /* compara cu 4*/
}
```

În urma execuției se va afișa:

Răspunsul este 1.

Întâi s-a evaluat expresia 2+1 și rezultatul s-a comparat cu 4, afișându-se valoarea logică a comparației care este 1 (adevărat).

În acest program apare un text cuprins între simbolurile /*...*/, care nu va fi luat în seamă de compilator și care formează ceea ce poartă denumirea de **comentariu**, util așa cum se știe la o mai bună înțelegere a programului. În cazul în care comentariul se întinde pe mai multe linii, este recomandabil (fără să fie obligatoriu) ca înaintea fiecărei linii de comentariu să se pună simbolul (*), ca în exemplul 17.18.

17.8. FUNCȚIA **getche ( )**

Înainte de a trece la probleme mai profunde de programare în limbajul C, să introducem această funcție specială de intrare **getche()** (get character with echo), care permite preluarea caracterelor imediat ce au fost tastate, fără a mai apăsa tasta „return". Caracterul tastat este afișat pe ecran în ecou. Astfel, spre deosebire de funcția **scanf()**, la care caracterul este asociat unei variabile menționată ca parametru al funcției, aici caracterul este asociat valorii funcției **getche()**.

Exemplul 17.19.

```
main()
{
    char ch;
    printf ("Tasteaza un caracter:");
    ch=getche();
    printf ("\n Caracterul tastat a fost: %c", ch);
}
```

La execuție are loc următoarea interacțiune cu operatorul:

```
Tasteaza un caracter: M
Caracterul tastat a fost: M
```

Primul caracter M (din primul rând) este introdus de la tastatură, fără a mai apăsa tasta „return".

EXERCIȚII

1. Care din aceste instrucțiuni în limbajul C sunt corecte ?

a) **int** a;

b) **char** b;

c) **double float** c;

d) **unsigned char** d.

2. Care din următoarele instrucțiuni inițializează o variabilă ?

a) num=2;

b) **int** num;

c) num<2;

d) **int** num=2.

3. Exprimați următoarele numere în notație zecimală:
a) 1.5e6;
b) 1.5e—6;
c) 7.6543e3;
d) —7.6543e—3.

4. Care din următorii sunt operatori aritmetici?
a) +
b) &
c) %
d) <

5. Rescrieți instrucțiunea următoare folosind operatorul de incrementare:
număr=număr+1

6. Rescrieți instrucțiunea următoare folosind operatorul de atribuire:
usa=usa+calif

7. Care este semnificația caracterelor '\t' și '\r'?

8. Funcția scanf() citește:
a) un singur caracter;
b) caractere și șiruri;
c) orice număr;
d) orice tip de variabilă.

9. Expresiile următoare sunt adevărate sau false?
a) 1>2;
b) 'a'<'b';
c) 1==2;
d) '2'=='2'.

10. Este corect făcut comentariul următor?

```
/* Acesta este
/* un comentariu care ocupa
/* mai multe linii
*/
```

11. Scrieți ce și cum tipărește următorul program:

```
main()
{
      int i, j, k;
      i=5; j=i%3; k=i++;
      printf ("%04d\n", i+=2);
      printf ("%0.4d\n",--j);
      printf ("%4.4d\n", j<=k);
}
```

12. Scrieți un program care să afișeze în zile și ore vârsta introdusă de la tastatură în ani.

13. Scrieți un program care să convertească și să afișeze în grade, în minute și în secunde un unghi dat în radiani. Unghiul este introdus de operator de la tastatură.

14. Scrieți un program care să calculeze și să afișeze suma cuburilor a două numere reale introduse de la tastatură.

15. Scrieți un program în care să introduceți dinainte stabilit un șir format din n caractere (n par < 80) și care să afișeze acest șir pe display, centrat pe linia respectivă (considerăm că pe o linie a display-ului încap cel mult 80 de caractere).

16. Scrieți un program care să permită introducerea unui caracter de la tastatură și care să afișeze caracterul introdus, codul lui ASCII și adresa (în hexa) la care a fost depus în memorie.

CAPITOLUL 18

STRUCTURI DE CONTROL ÎN LIMBAJUL C

În limbajul C **structurile de control** sunt de două feluri:
— *structuri iterative* (*bucle*);
— *structuri decizionale.*

18.1. STRUCTURI ITERATIVE (BUCLE)

Iteraţia este o operaţie frecvent întâlnită în activitatea de programare. În limbajul C există trei **structuri de bază** care permit programarea operaţiilor iterative:
— *bucla FOR*
— *bucla WHILE*
— *bucla DO WHILE*

18.1.1. Bucla FOR

Reprezintă structura care se recomandă pentru programarea unor operaţii care se repetă de un număr dat (fix) de ori.

Exemplul 18.1.

```
/*bucla for.c*/
/*afişează numerele de la 0 la 9*/
main()
{
        int num;
        for(num=0; num<10; num++)
                printf ("num= %d\n", num);
}
```

În ipoteza că programul este compilat şi transformat într-un program executabil cu denumirea bucla_for, la execuţie pe display va apărea:

```
C:\<bucla_for
num=1
num=2
   :
   :
   :
num=9
```

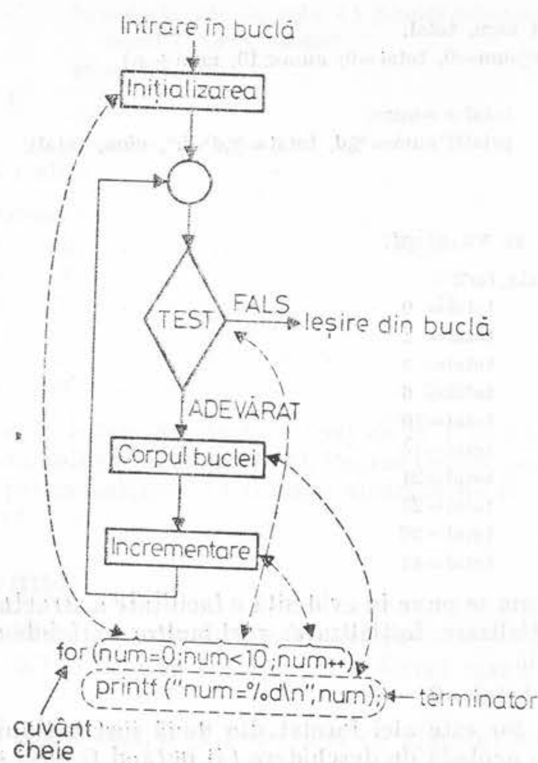


Fig. 18.1.

Cu ajutorul acestui exemplu, să precizăm structura unei bucle for- figura 18.1.

Variabila *num* este utilizată pentru controlul operațiilor din buclă. În acest exemplu ea a fost incrementată (cu pas 1) în partea a treia a buclei, dar putea la fel de bine să fie incrementată cu orice alt pas sau să fie decrementată. De exemplu, se putea scrie în loc de num++:

num=num+3 sau

num-- sau

num=num-5

Inițializarea variabilei *num* s-a făcut aici cu valoarea 0, dar se putea face cu orice altă valoare. Aici se poate vedea mai clar modul de lucru al acestei structuri. Întâi se face inițializarea variabilei sau expresiei de lucru. Apoi se testează condiția care determină executarea corpului buclei (dacă condiția este adevărată) sau se iese din buclă dacă condiția este falsă. După fiecare execuție a corpului buclei se incrementează (sau se decrementează) variabila sau expresia de lucru, revenindu-se apoi la test.

În cazul în care corpul buclei **for** este compus din mai multe instrucțiuni, ele vor forma un **bloc** cuprins între paranteze acolade.

Exemplul 18.2.

```
/*bucla_for2.c*/
/*tipărește numerele naturale de la 0 la 9 și suma lor progresivă*/
main()
{
```

```
    int num, total;
    for(num=0, total=0; num<10; num++)
    {
        total+=num;
        printf("num= %d, total= %d\n", num, total);
    }
}
```

În urma execuţiei se va afişa:

```
c:\>bucla_for2
num=0,    total= 0
num=1,    total= 1
num=2,    total= 3
num=3,    total= 6
num=4,    total=10
num=5,    total=15
num=6,    total=21
num=7,    total=28
num=8,    total=36
num=9,    total=45
```

În acest program se pune în evidenţă o facilitate a structurii, de a permite în secvenţă de iniţializare, iniţializarea mai multor variabile sau expresii despărţite prin virgulă.

num=0, total=0

Corpul buclei **for** este aici format din două instrucţiuni, care formează un **bloc**, paranteza acoladă de deschidere ({) putând fi pusă şi în linie cu expresia buclei **for:**

```
for(num=0, total=0; num< 10; num++) {
    total+=num;
    printf ("num= %d, total= %d/n", num, total);
}
```

Putem face acum precizarea că structura generală a buclei **for** este

```
for (expr.1; expr.2; expr.3)
{
bloc
}
```

Oricare din cele trei expresii pot lipsi, iar dacă expr. 2 lipseşte este considerată adevărată.

Bucle for incluse una în cealaltă

Să analizăm următorul exemplu care afişează *tabla înmulţirii cu numere de la 1 la 14*:

Exemplul 13.3.

```
/*tabmul.c*/
/*generează tabla înmulţirii*/*
main()
{
    int col, lin;
    for(lin=1; lin<15; lin++) /*bucla exterioară*/
    {
```

```
            for(col=1; col<15; col++) /*bucla interioară*/
                printf ("%4d", col*lin);
            printf ("\n");
        }
}
```

La execuţie se va afişa:

```
c:\>tabmul
1   2  . . . . . . . . . . . . .  14
2   4  . . . . . . . . . . . . .  28
3   6  . . . . . . . . . . . . .  42
:   :                              :
:   :                              :
14  28 . . . . . . . . . . . . . 196
```

Bucla interioară baleiază coloanele (*col*) de la 1 la 14, în timp ce bucla exterioară baleiază liniile (*lin*) de la 1 la 14. Pentru fiecare linie bucla interioară care se execută prima baleiază 14 coloane tipărind de fiecare dată valoarea produsului lin*col.

18.1.2. Bucla WHILE

O altă modalitate de programare a unei operaţii iterative o constituie utilizarea buclei WHILE, care are aparent o formă mai simplă decât bucla FOR.

Să luăm exemplul 18.2 şi să-l rescriem cu ajutorul buclei **while**.

Exemplul 18.4.

```
/*bucla while.c*/
/*tipăreşte numerele naturale de la 0 la 9 şi suma lor progresivă utilizînd bucla
while*/
main()
{
        int num=0;
        int total=0;
        while (num < 10)
        {
            total+=num;
            printf ("num=%d, total=%d\n", num++, total);
        }
}
```

Execuţia acestui program produce tabelul din exemplul 18.2. Schema logică de lucru a buclei **while** este similară cu cea a buclei **for** şi este prezentată în figura 18.2.

Se observă că şi în acest caz testul este la început. Iniţializarea se face însă în afara buclei **while** (se face la declararea tipului de variabilă). Incrementarea variabilei se face în corpul buclei cu ajutorul operatorului (++).

Este bine să facem precizarea că *în cazul în care numărul de iteraţii este cunoscut, bucla* **while** *şi bucla* **for** *sunt similare, dar se preferă bucla* **for** *datorită construcţiei sale mai clare.*

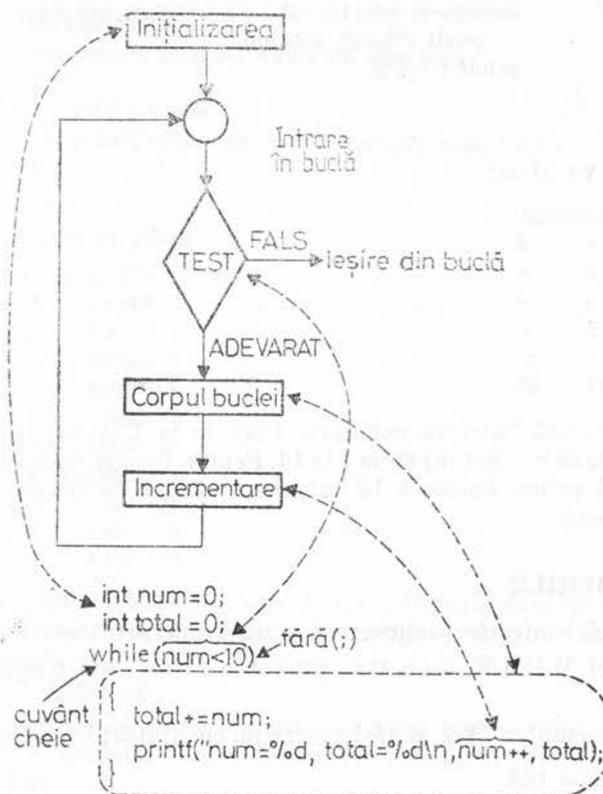


Fig. 18.2.

Să considerăm exemplul următor care permite **numărarea și afișarea** caracterelor unui text.

Exemplul 18.5.

```
/*num_car .c*/
/*numără şi tipăreşte caracterele unui text*/
main()
{
        int num=0;
        printf ("Scrieţi un text:\n");
        while(getche() !='\r')
              num++;
        printf ("Numărul de caractere este %d\n", num);
}
```

Programul vă invită să scriem un text care se **consideră încheiat cînd** se apasă pe tasta [Return]. Până atunci programul va **rămâne în bucla while**, va preluа fiecare caracter introdus de la tastatură și **va calcula numărul** acestor caractere. La apăsarea tastei [Return] se va ieși din **buclă și** se va afișa numărul acestor caractere din textul introdus.

O execuţie a acestui program s-ar putea desfăşura astfel:

```
C:\>num_car
Scrieţi un text:
Noi suntem elevi
Numărul de caractere este 16
```

Se observă că în acest program nu există nici o variabilă de buclare; nu se face nici iniţializarea şi nici incrementarea unei astfel de variabile. Condiţia de menţinere în buclă sau de ieşire din ea este dată de valoarea logică a expresiei:

(getche() != '\r')

Funcţia **getche()**, aşa cum am arătat în capitolul anterior, returnează în programul apelant instantaneu valoarea caracterului tastat. Acest caracter va fi comparat cu *caracterul „return"* reprezentat prin *secvenţa „escape"* '\r'. Dacă se apasă orice altă tastă în afara tastei [Return], programul va rămâne în buclă incrementînd variabila *num*, care contorizează astfel numărul de caractere introduse. La apăsarea tastei [Return] se iese din buclă (condiţia din paranteză devine falsă) şi se va afişa valoarea variabilei *num* (fig. 18.3).

getche( ) are valoarea caracterului tastat
Programul va rămâne în buclă atâta timp cât acest caracter este diferit de acest caracter

```
while (getche() != '\r')
```

condiţia logică (test) (adevărată sau falsă)

Fig. 18.3.

În acest exemplu se poate observa prezenţa funcţiei **getche** ( ) în expresia condiţiei logice a buclei **getche** ceea ce la prima vedere ar putea constitui o soluţie mai puţin uzuală. Programul se mai poate scrie scoţând funcţia în afara acestei expresii dar programul ar rezulta mai complicat.

Bucle infinite

Expresiile utilizate în condiţia logică a unei bucle **while** pot fi relativ complexe, dar ele pot fi şi foarte simple, ca în exemplul următor care tipăreşte codurile ASCII ale caracterelor introduse de la tastatură:

Exemplul 18.6.

```
/*car_ascii.c*/
/*tipăreşte codul ASCII al caracterelor*/
main()
{
    while (1)
    {
        char ch;
        printf ("\nIntroduceţi caracterul:\n");
        ch=getche();
        printf ("\nCodul lui %c este %d\n", ch,ch);
    }
}
```

Programul cere utilizatorului să tasteze un caracter şi va afişa caracterul şi codul lui ASCII.

O execuţie a acestui program ar putea fi:

```
C:\< car_ascii
Introduceţi caracterul:
a
Codul lui a este 97.
Introduceţi caracterul:
A
Codul lui A este 65
```

Particularitatea acestui program constă în faptul că el se află într-o *buclă infinită*. El nu se termină, putând fi executat ori de câte ori se doreşte fără a fi lansat de fiecare dată în execuţie. Condiţia logică a buclei **while** are permanent valoarea 1 şi deci este întotdeauna adevărată. Pentru ieşirea din acest program (pentru terminarea lui) se folosesc diferite procedee. Astfel, în sistemul de operare MS-DOS combinaţia de taste (apăsate simultan) [Ctrl] [C] va permite terminarea programului şi revenirea în sistemul de operare.

Acest program putea fi realizat şi cu ajutorul construcţiei **for** punând:

```
for(; ;)
```

în loc de

```
while(1)
```

Bucle while cuprinse una în alta

Ca şi în cazul buclelor **for**, buclele **while** pot fi incluse una în cealaltă, aşa cum rezultă din exemplul următor:

Exemplul 18.7.

```
/*joc .c*/
/*ghiciţi o literă*/
main()
{
    char ch;
    while (1)
    {
        printf ("\nTastaţi o literă între 'a' şi 'm':\n");
        while ((ch=getche())!='e')
        {
            printf ("\n%c nu este corect\n", ch);
            printf ("\n Încercaţi din nou\n");
        }
        printf ("Aceasta este litera corectă!\n");
    }
}
```

Programul cere utilizatorului să tasteze un caracter cuprins între 'a' şi 'm' (litere mici). Atâta timp cât acest caracter, preluat de funcţia **getche()** şi atribuit variabilei ch este diferit de 'e' programul rămâne în bucla **while** interioară afişînd că nu este corect caracterul introdus şi cerând utilizatorului să încerce din nou. Dacă de la tastatură se va introduce caracterul 'e', programul va ieşi din bucla interioară rămânând în bucla exterioară infinită afişând că este corect caracterul introdus şi cerând utilizatorului intro-

ducerea unui nou caracter între 'a' și 'm'. Ieșirea din această buclă și terminarea programului se poate face de exemplu tot cu combinația de taste [Ctrl] [C].

O execuție a programului ar putea fi:

```
C:\>joc
Tastați o literă între 'a' și 'm':
a
a nu este corect
Încercați din nou
k
k nu este corect
Încercați din nou
e
Aceasta este litera corectă
Tastați o literă între 'a' și 'm'.
^C
```

În acest exemplu se observă că unul din cei doi operanzi ai condiției logice aferente buclei **while** este format dintr-o expresie care va căpăta o anumită valoare — figura 18.4.

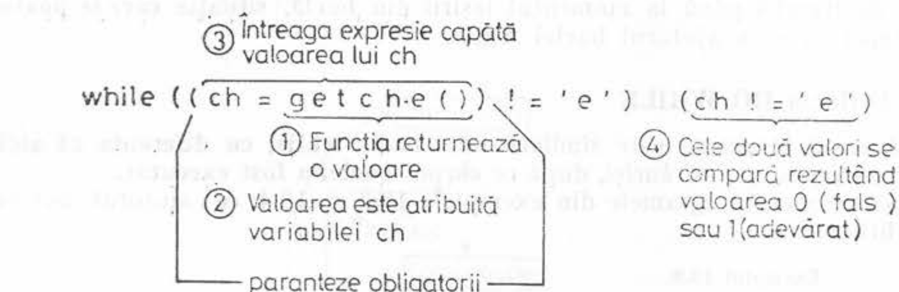


Fig. 18.4.

Dacă expresia din stânga operatorului condițional (!=) nu ar fi pusă între paranteze, adică s-ar fi scris

while (ch=getche()!='e')

s-ar fi semnalat eroare deoarece ordinea operațiilor ar fi fost următoarea (vezi tabelul cu precedența și asociativitatea operatorilor de la § 18.2): întâi se execută operatorul relațional și apoi operatorul de atribuire. Rezultatul comparației ar fi 0 sau 1 și această valoare s-ar atribui lui ch.

Să mai considerăm un exemplu, de *program care calculează și afișează factorul uni număr natural.*

Exemplul 18.8

```
/*factorial .c*/
/*calculează factorialul unui număr natural*/
main()
{
        long int num, fact;
        while(1)
        {
                printf ("\nIntroduceți numărul!");
```

```
        scanf ("%ld", &num);
        fact =1;
        while (num> 1)
              fact*=num— —;
        printf ("Rezultatul este: %ld", fact);
    }
}
```

Poate avea loc următoarea execuție a programului:

```
C:\>factorial
Introduceți numărul ! 3
Rezultatul este 6
Introduceți numărul! 5
Rezultatul este 120
^C
```

Ieșirea din program se face prin comanda de la tastatură, cu combinația de taste [Ctrl] [C] (în MS_DOS).

Din cauza numerelor mari la care se poate ajunge, este indicat aici să se lucreze cu variabile **„long int"**.

Se observă de asemenea în acest exemplu necunoașterea apriori a numărului de iterații până la momentul ieșirii din buclă, situație care se poate programa ușor cu ajutorul buclei **while**.

18.1.3. Bucla DO WHILE

Această structură este similară unei bucle **while**, cu diferența că aici *testul are loc la sfîrșitul buclei*, după ce corpul buclei a fost executat.

Să rescriem programele din exemplele 18.2 și 18.4 cu ajutorul buclei **do while**.

Exemplul 18.9.

```
/*Bucla_ do_while.c*/
/*tipărește numerele naturale de la 0 la 9 și suma lor progresivă utilizând bucla
do while */
main()
{
        int num=0;
        int total=0;
        do
        {
              total+=num;
              printf ("num= %d, total= %d", num++, total);
        }
        while(num < 10);
}
```

Execuția programului va genera tabelul:

```
num=0,     total=0
num=1,     total=1
num=2,     total=3
num=3,     total=6
num=4,     total=10
num=5,     total=15
```

```
num=6,    total=21
num=7,    total=28
num=8,    total=36
num=9,    total=45
```

acelaşi ca în exemplele 18.2 şi 18.4.

Se observă că această structură beneficiază de două cuvinte cheie: cuvântul **do**, care marchează începutul buclei (nu are alt rol) şi cuvîntul **while**, care marchează sfârşitul buclei şi conţine testul buclei. Foarte important de reţinut faptul că bucla se termină cu (;).

Schema de lucru a buclei **do while** este prezentată în figura 18.5.

O particularitate a acestei structuri constă în faptul că bucla se va executa cel puţin odată, chiar dacă testul nu este îndeplinit de la început.

Există numeroase situaţii de acest fel care trebuie programate. Să reluăm exemplul 18.7 şi să-l completăm în acest sens.

Exemplul 18.10.

```
/*joc1.c*/
/*ghiciţi o literă*/
main()
{
     char ch;
     do
     {
```

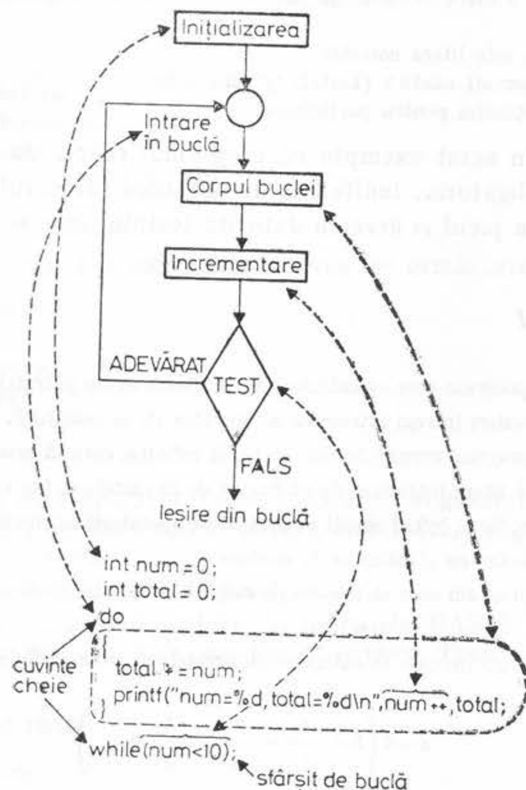


Fig. 18.5.

```
        printf ("Tastaţi o literă între 'a' şi 'm':\n");
        while ( (ch=getche())!='e')
        {
            printf ("\n%c este incorect\n", ch);
            printf ("\nÎncercaţi din nou.\n");
        }
        printf ("\nAceasta este litera corectă!\n");
        printf ("\nMai încercaţi o dată? (Tastaţi 'y' sau 'n'):");
    }
    while (getche()=='y');
    printf ("Va multumim pentru participare!\n");
}
```

Rezultă următoarea sesiune de lucru:

```
C:\>joc 1
Tastaţi o literă ,între 'a' şi 'm':
b
b este incorect
Încercaţi din nou
e
Aceasta este litera corectă!
Mai încercaţi o dată? (Tastaţi 'y' sau 'n'):y
Tastaţi o literă între 'a' şi 'm':
e
Aceasta este litera corectă!
Mai încercaţi odată? (Tastaţi 'y' sau 'n'):n
Vă mulţumim pentru participare!
```

Se observă în acest exemplu că programul (bucla **do while**) se execută odată în mod obligatoriu, indiferent de opţiunea jucătorului de a abandona sau de a continua jocul şi aceasta datorită testului care se face la sfârşit.

EXERCIŢII

1. Scrieţi un program care să calculeze şi să afişeze suma pătratelor numerelor naturale cuprinse între două valori întregi introduse de operator de la tastatură.

2. Scrieţi un program care să lucreze în buclă infinită, care să calculeze şi să afişeze câte caractere separă două litere introduse de utilizator de la tastatură (de exemplu între 'a' şi 'd' există două caractere, 'b' şi 'c'). Folosiţi avantajul că operatorii aritmetici lucrează asupra variabilelor de tip caracter ca şi când ar fi numere.

3. Scrieţi un program care să traseze pe display două linii de câte n (*) ; n se introduce de la tastatură.

4. Scrieţi un program care să calculeze cu precizia ε valoarea lui π folosind seria:

$$x=4\left(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\ldots\right)$$

Valoarea lui ε se introduce de la tastatură.

Considerăm că numărul π este calculat cu precizia dorită ε dacă $|\pi_{n+1}-\pi_n| \leqslant \varepsilon$ unde π_{n+1} şi π_n sunt două aproximaţii succesive ale lui π.

5. Scrieţi un program care să calculeze şi să afişeze valoarea polinomului $9x^3-2x^2+120x-130$ în puncte care se introduc de la tastatură. Numărul de puncte nu este cunoscut apriori.

18.2. STRUCTURI DECIZIONALE

Limbajul C conţine **trei structuri majore** pentru rezolvarea problemelor decizionale: *structura if*, *structura if-else* şi *structura switch*. La acestea se mai adaugă încă două posibilităţi mai rar utilizate: *operatorul condiţional* şi *instrucţiunea goto.*

18.2.1. Structura if

Limbajul C utilizează, ca multe alte limbaje de programare, **cuvântul cheie if** pentru realizarea construcţiei decizionale. Să dăm un exemplu simplu pentru a vedea despre ce este vorba.

Exemplul 18.11.

```
/*test_if .c*/
/*arată modul de utilizare al instrucţiunii if*/
main()
{
        char ch;
        ch=getche();
        if(ch=='y')
                printf ("Aţi tastat y\n");
}
```

La lansarea programului în execuţie pot avea loc următoarele situaţii:

```
C:\>testif
y
Aţi tastat y
C:\>testif
n
C:\>
```

Se observă că dacă operatorul apasă tasta y, programul răspunde cu mesajul „Aţi tastat y", iar dacă apasă oricare altă tastă, programul se termină fără să mai afişeze ceva.

În figura 18.6 este detaliată structura **if.**

Să remarcăm că, spre deosebire de limbajele BASIC sau Pascal, *în C cuvântul cheie* **if** *nu este urmat de cuvântul cheie* **then. Then** *nu este cuvânt cheie în limbajul C.*

Exemplul 18.12.

```
/*count .c*
/*numără caracterele şi cuvintele introduse de la tastatura*/
main()
```

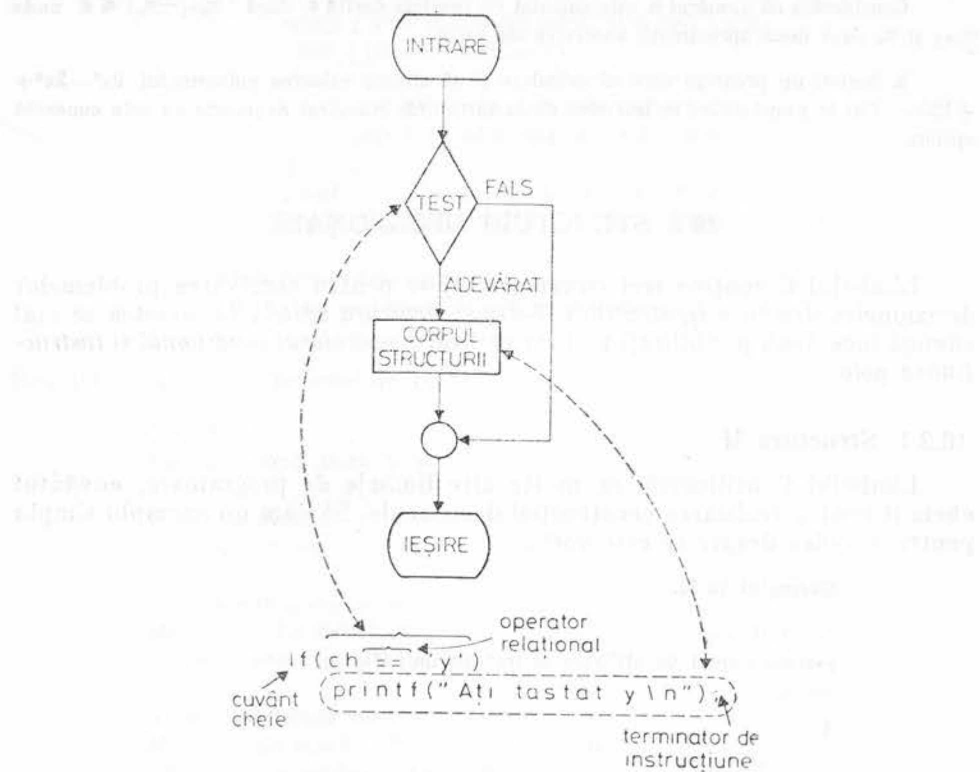


Fig. 18.6.

```
{
        int car=0;
        int cuv=0;
        char ch;
        printf („Scrieţi textul\n");
        while((ch=getche())!='\r')        /*citeşte caractere de la tastatură până
        {                                 /*se tastează <CR>*/
                car++;                    /*numără caracterele*/
                if (ch==' ')
                        cuv++;            /*numără cuvintele*/
        }
        printf ("Numărul de caractere este %d\n", car);
        printf ("Numărul de cuvinte este %d\n", cuv+1);
}
```

În program, numărul de cuvinte este egal cu numărul de blancuri (care se numără) plus 1.
Execuţia programului poate fi:

```
C:\>count
Scrieţi textul
Noi suntem elevi
Numărul de caractere este 16
Numărul de cuvinte este 3
```

● **Instrucţiuni multiple în corpul instrucţiunii if.** Ca şi în cazul buclelor, în corpul instrucţiunii **if** pot apare mai multe instrucţiuni, ele trebuind să formeze în acest caz un **bloc** cuprins între paranteze acolade.

Exemplul 18.13.

```
/*test_if2 .c* /
/*arată cum pot apare mai multe instrucţiuni în corpul unei structuri if*/
main()
{
        char c;
        ch=getche();
        if(ch=='y')
        {
                printf ("Aţi tastat");
                printf ("caracterul %c\n", ch);
        }
}
```

Execuţia programului poate fi:

```
c:\>test_if2
y
Aţi tastat caracterul y
```

● **Construcţii if cuprinse una în corpul celeilalte.** O astfel de situaţie este prezentată în exemplul următor:

Exemplul 18.14.

```
/*nested_if .c* /
/*structuri if cuprinse una în corpul celeilalte* /
main()
{
        if (getche()=='o' )
                if (getche()=='m')
                        printf ("Aţi tastat om\n");
}
```

Se observă aici că cel de-al doilea **if** este inclus în corpul primului **if** şi el se va executa numai dacă condiţia logică din structura primului **if** este adevărată. Instrucţiunea **printf** se va executa numai dacă ambele condiţii logice ale celor două construcţii **if** sunt adevărate.
În urma execuţiei putem avea:

```
C:\>nested_if
n
C:\>nested_if
on
C:\>nested_if
om
Aţi tastat om
C:\>
```

Schema logică a instrucţiunilor **if** cuprinse una în cealaltă este prezentată în figura 18.7.

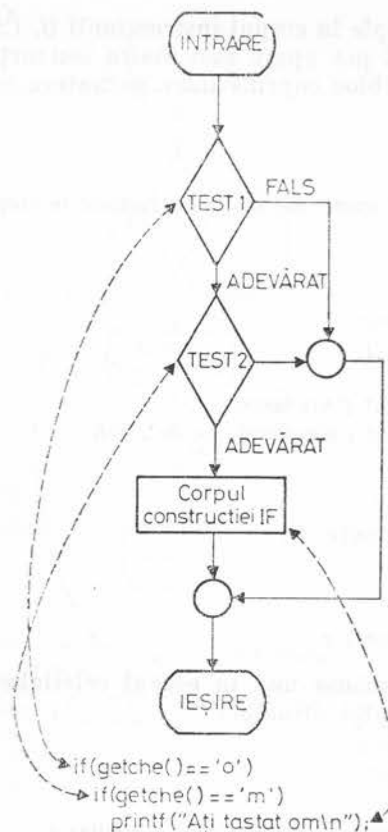


Fig. 18.7.

18.2.2. Structura if-else

Structura **if** anterioară determină executarea unei instrucțiuni sau a unui grup de instrucțiuni, dacă rezultatul testului era adevărat și nu specifica nimic în cazul în care rezultatul testului era fals. În acest caz există alternativă și pentru rezultatul fals.

Exemplul 18.15.

```
/*test_else .c*/
/*ilustrează modul de lucru al construcției if_else*/
main()
{
        char ch;
        ch=getche();
        if(ch=='y')
                printf ("Ați tastat y\n");
        else
                printf ("Nu ați tastat y\n")
}
```

O posibilă execuţie a acestui program ar fi:

```
C:\>test_else
y
Aţi tastat y
C:\>test_else
n
Nu aţi tastat y
```

Schema de lucru a acestei construcţii este arătată în figura 18.8.

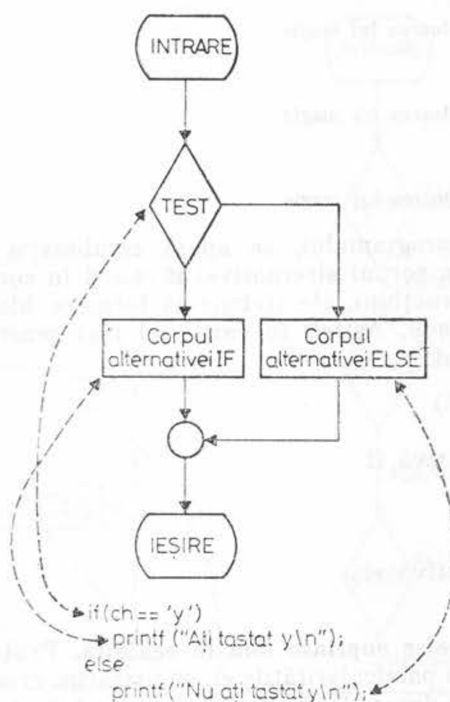


Fig. 18.8.

Să mai considerăm un exemplu care să ilustreze şi mai bine modul de lucru al construcţiei **if-else**.

Exemplul 18.16.

```
/*magic .c* /
/*ghicirea unui număr întreg generat de o funcţie* /
#include <stdlib.h>
main()
{
        int magic;
        int num;
        while(1)
        {
                magic=rand();
                printf ("Ghiceşte valoarea lui magic\n");
                scanf ("%d", &num);
```

```
        if(num= =magic)
                printf ("**Corect**\n");
        else
                printf ("**Incorect**!\n");
        }
}
```

Funcția rand() returnează un număr întreg aleator între 0 ÷ 32.767. Programul lucrează în bucla infinită și poate conduce la următoarea execuție:

```
c:\< magic
Ghicește valoarea lui magic
1250
Incorect
Ghicește valoarea lui magic
1275
Corect
Ghicește valoarea lui magic
```

Pentru terminarea programului, se apasă combinația de taste [Ctr] [C]. În cazul în care și în corpul alternativei **if** sau/și în corpul alternativei **else** apar mai multe instrucțiuni, ele trebuie să formeze blocuri fiind cuprinse între paranteze acolade. Astfel, în cazul cel mai general, structura **if-else** are următorul conținut:

```
if(condiție logică)
{
    bloc alternativă if
}
else
{
    bloc alternativă else
}
```

● **Construcții if-else cuprinse una în cealaltă.** Pentru a ilustra această posibilitate precum și particularitățile ei, să urmărim următorul program care răspunde cu aprecieri despre timpul atmosferic pe baza temperaturii mediului introdusă de operator de la tastatură.

Exemplul 18.17.

```
/*temper.c* /
/*apreciază starea timpului atmosferic* /
main()
{
        int temp;
        printf ("Introduceți temperatura mediului:");
        scanf (" %d", &temp) ;
        if (temp <25)
                if (temp >20)
                        printf ("Este o zi frumoasă\n");
                else
                        printf ("Este răcoare\n");
        else
                printf ("Este prea cald\n");
}
```

Lansarea în execuţie a programului poate duce la următoarea situaţie:

```
C:\>temper
Introduceţi temperatura mediului : 22
Este o zi frumoasă.
```

Diagrama logică a acestei construcţii pentru exemplul de mai sus este arătată în figura 18.9.

În cazul a n construcţii **if-else** cuprinse una în cealaltă, diagrama logică este arătată în figura 18.10.

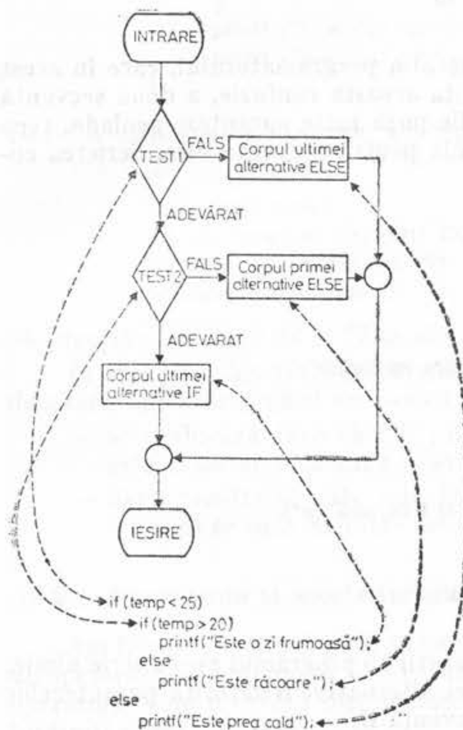


Fig. 18.9.

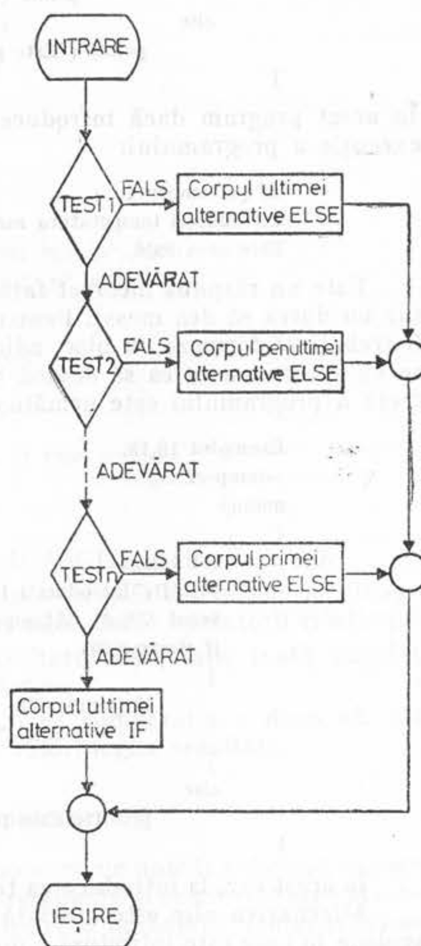


Fig. 18.10.

La realizarea unor astfel de construcţii, trebuie avută în vedere regula de asociere a alternativelor **else** cu alternativele **if**. Astfel, un **else** este asociat cu cel mai apropiat **if** care nu are propriul lui **else** asociat.

Să considerăm din noul program din exemplul 18.17, la care să-i eliminăm prima alternativă **else**. Ea este asociată în mod logic cu cea de a doua alternativă **if**. Prin dispariţia ei însă, automat, alternativa **else** rămasă va fi asociată cu cea de a doua alternativă **if** şi nu cu prima, aşa cum am dori noi.

Exemplul 18.18.

```
/*temper 1.c*/
main()
{
        int temp;
        printf ("Introduceţi temperatura mediului:");
        scanf ("%d", &temp);
        if (temp< 25)
                if (temp>20)
                        printf ("Este o zi frumoasa\n");
        else
                printf ("Este prea cald\n");
}
```

În acest program dacă introducem temperatura 15 va rezulta următoarea execuţie a programului:

```
C:\<temper 1.
Introduceţi temperatura mediului: 15
Este prea cald.
```

Este un răspuns incorect faţă de intenţia programatorului, care în acest caz nu dorea să dea mesaj. Pentru a evita această confuzie, a doua secvenţă **if** trebuie să formeze un bloc, adică să fie pusă între paranteze acolade, ceea ce va determina ca ea să devină invizibilă pentru secvenţa **else**. Scrierea corectă a programului este următoarea:

Exemplul 18.19.

```
/*temper2.c*/
main()
{
        int temp;
        printf ("Introduceţi temperatura mediului:");
        scanf ("%d", &temper);
        if (temp< 25)
        {
                if (temp>20)
                        printf ("Este o zi frumoasă\n");
        }
        else
                printf ("Este prea cald\n");
}
```

În acest caz, la introducerea temperaturii 15 programul nu va scrie nimic.

Alternativa **else** este asociată primei alternative **if** datorită parantezelor acolade în care este introdusă a doua secvenţă **if**.

18.2.3. Operatori logici

În cazul construcţiilor **if-else** complicate, în limbajul C există un mod eficient şi clar de scriere a expresiilor logice, utilizând **operatori logici**. Există trei astfel de operatori:

|| *SAU logic (SAU INCLUSIV)*
&& *ŞI logic*
! *NU logic*

Pentru a ilustra modul de folosire a acestor operatori, să considerăm următorul program care numără caracterele și cifrele dintr-un text.

Exemplul 18.20.

```
/*num_car_cif .c* /
/*numără caracterele şi cifrele dintr-un text*/
main()
{
        int car=0;
        int cif=0;
        char ch;
        printf ("Introduceţi textul\n");
        while((ch=getche())!='\r')
        {
              car++;
              if(ch>47 & & ch<58)
                    cif++;
        }
        printf ("Numărul caracterelor este %d\n", car);
        printf ("Numărul cifrelor este %d\n", cif);
}
```

Rezultă următoarea execuție a programului:

```
C:\>num_car_cif
Introduceţi textul
Eu am cumpărat 100 cărţi 250 caiete şi 1300 creioane
Numărul caracterelor este 53
Numărul cifrelor este 10
```

Să precizăm ca între 48 și 57 se află codurile ASCII ale cifrelor 0÷9.

În expresia ch>47 && ch<58 evaluarea începe de la stînga la dreapta deoarece operatorii logici au prioritate mai mică decât operatorii relaţionali:

- se evaluează dacă **ch<47** ; dacă rezultatul este fals, toată expresia este falsă și încetează operațiile logice;
- dacă rezultatul este adevărat, se face comparaţia a doua **ch<58**, iar apoi se face **SI** între cele două valori logice rezultate.

18.2.4. Precedența și asociativitatea operatorilor

Am făcut cunoștință până în acest moment cu un număr suficient de mare de operatori. Se impune să vorbim despre precedența și asociativitatea acestor operatori. Vom prezenta toți operatorii utilizați în limbajul C (chiar și cei neîntâlniți până acum).

Precedența operatorilor ne spune cum se evaluează expresiile care conţin doi sau mai mulți operatori, aplicați unui același operand sau unor operanzi diferiţi, în situaţia când nu există paranteze care să impună o anumită ordine a operațiilor. De exemplu, în expresia:

!a&&b

apar doi *operatori*: cel *de negare* și cel *de conjuncție logică*. Întrebarea este ce operator „leagă" mai tare, ! sau &&? În acest caz, răspunsul este următorul: expresia se evaluează ca și când ar fi scrisă:

(!a)&&b

Asociativitatea operatorilor ne spune cum se evaluează expresiile în care apare un acelaşi operator, de mai multe ori. De exemplu, în expresia:

a =x/y/z;

apar două împărţiri. Expresia se poate evalua ca x/y şi apoi rezultatul se împarte la z, sau se împarte y la z şi apoi se împarte x la această valoare.

Operatorul de împărţire are asociativitate de la stânga la dreapta, adică expresia se interpretează ca şi cum ar fi fost scrisă:

a =(x/y)/z;

Operatorul de atribuire are asociativitate de la dreapta la stînga, deci expresia:

a =b =c =d;

se evaluează ca şi cum ar fi scrisă:

a =(b =(c =d));

Asociativitatea poate fi modificată prin paranteze, de exemplu:

a =x/(y/z);

Precedenţa şi asociativitatea operatorilor definesc modul de evaluare a unei expresii nu şi ordinea temporală în care se evaluează operanzii. Ordinea evalurării operanzilor într-o expresie este garantată numai la operatorii && (Şi logic), || (sau logic),?: (condiţional) şi, (virgula).

Operatorii limbajului C se împart în *15 categorii de precedenţă*, prezentate în tabel, în ordine descrescătoare. Categoria 1 are precedenţa cea mai mare („leagă" cel mai puternic). Operatorul virgulă are precedenţa cea mai mică („leagă" cel mai slab). Operatorii din aceeaşi categorie au aceeaşi precedenţă.

Operatorii unari (categoria 2), cel condiţional (categoria 13) şi cei de atribuire (categoria 14) se asociază de la dreapta la stânga. Toţi ceilalţi operatori se asociază de la stânga la dreapta.

Categorie	Operatori	Semnificaţie
1. Prioritate maximă	() [] -> .	Apel de funcţie Expresie cu indici Selectori de membru la structuri
2. Operatori unari	! ~ + — ++ —— & * sizeof (tip)	Negare logică Negare bit cu bit (1's complement) Plus şi minus unari Incrementare/decrementare (pre şi post) Luarea adresei Indirectare Dimensiune operand (in octeţi) Conversie explicită de tip
3. Operatori de multiplicare	* / %	Înmulţire, împărţire, rest
4. Adunare, scădere	+ —	Plus şi minus binari
5. Deplasări	<< >>	Deplasare stânga sau dreapta
6. Relaţionali	< <= > >=	Mai mic, mai mic sau egal, mai mare, mai mare sau egal
7. Egalitate	== !=	Egal, diferit

Categorie	Operatori	Semnificaţie
8.	&	SI logic bit cu bit
9.	∧	SAU EXCLUSIV bit cu bit
10.	|	SAU logic bit cu bit
11.	&&	SI logic
12.	||	SAU logic
13. Operatorul condiţional	? :	Operatorul condiţional (ternar)
14. Operatori de atribuire	=	Atribuire simplă
	*= /= %=	Atribuire produs, cât, rest
	+= −=	Atribuire sumă, diferenţă
	&= ∧= |=	Atribuire şi, sau exclusiv, sau (bit)
	<<= >>=	Atribuire deplasare stânga/dreapta
15. Virgula	,	Evaluare op1, op2. Valoarea expresiei este op2

18.2.5. Construcţia else-if

Pînă acum am arătat cum pot fi incluse una în cealaltă construcţiile **if-else**. Să vedem în continuare o serie de exemple mai complexe privind acest aranjament.

Exemplul 18.21.

```
/*calcul.c*/
/*efectuează cele patru operaţii aritmetice*/
main()
{
      float num1, num2;
      char op;
      while(1)
      {
            printf ("Tastează număr, operator, număr:\n");
            scanf ("%f %c %f", &num1, &op, &num2);
            if(op=='+')
                  printf ("%f\n", num1+num2);
            else
                  if(op=='−')
                        printf ("%f\n", num1−num2);
                  else
                        if(op=='*')
                              printf ("%f\n", num1*num2);
                        else
                              if(op=='/')
                                    printf ("%f\n", num1/num2);
            printf ("\n\n");
      }
}
```

Programul permite efectuarea celor patru operaţii cu două numere reale. El cere operatorului să introducă primul număr, operatorul aritmetic şi numărul al doilea, furnizînd la ieşire rezultatul operaţiei.
Poate rezulta următoarea execuţie :

```
C:\<calcul
Tastează număr, operator, număr:
5 * 3
15.000000
Tastează număr, operator, număr:
. . . . . . . . . . . . . . . .
```

Se iese din program (din bucla infinită) cu ajutorul combinaţiei de taste [Ctrl] [C].

Important în acest program este modul în care sunt incluse una în alta structurile **if-else**. Fiind multe, programul scris în această manieră este greu de citit. Ca atare se recomandă un alt mod de scriere a lui, ceea ce va da naştere aparent la o nouă structură numită **else-if**. Evident, aceasta este doar o pseudostructură, generată doar de modul de scriere a programelor cu multe structuri **if-else**, cuprinse una în alta. Să urmărim în continuare acelaşi program de mai sus, dar altfel scris.

Exemplul 18.22.

```
/*calcul1.c*/
/*efectuează cele patru operaţii aritmetice*/
main()
{
        float num1, num2;
        char op;
        while(1)
        {
                printf ("Tastează număr, operator, număr:\n");
                scanf ("%f %c %f", &num1, &op, &num2);
                if(op=='+')
                        printf ("%f\n", num1+num2);
                else if(op=='—')
                        printf ("%f\n", num1—num2);
                else if(op=='*')
                        printf ("%f\n", num1*num2);
                else if(op=='/')
                        printf ("%f\n", num1/num2);
                printf ("\n\n");
        }
}
```

Cele două programe sunt identice din punct de vedere al acţiunii lor, dar acesta din urmă este mai uşor de citit. În această nouă construcţie **else-if**, dacă condiţia logică este adevărată, se execută corpul structurii **else-if** (în acest caz **printf()**) şi se sare la sfârşitul lanţului **else-if**. În caz contrar, se trece la următoarea construcţie **else-if**. Schema logică asociată acestei construcţii este prezentată în figura 18.11.

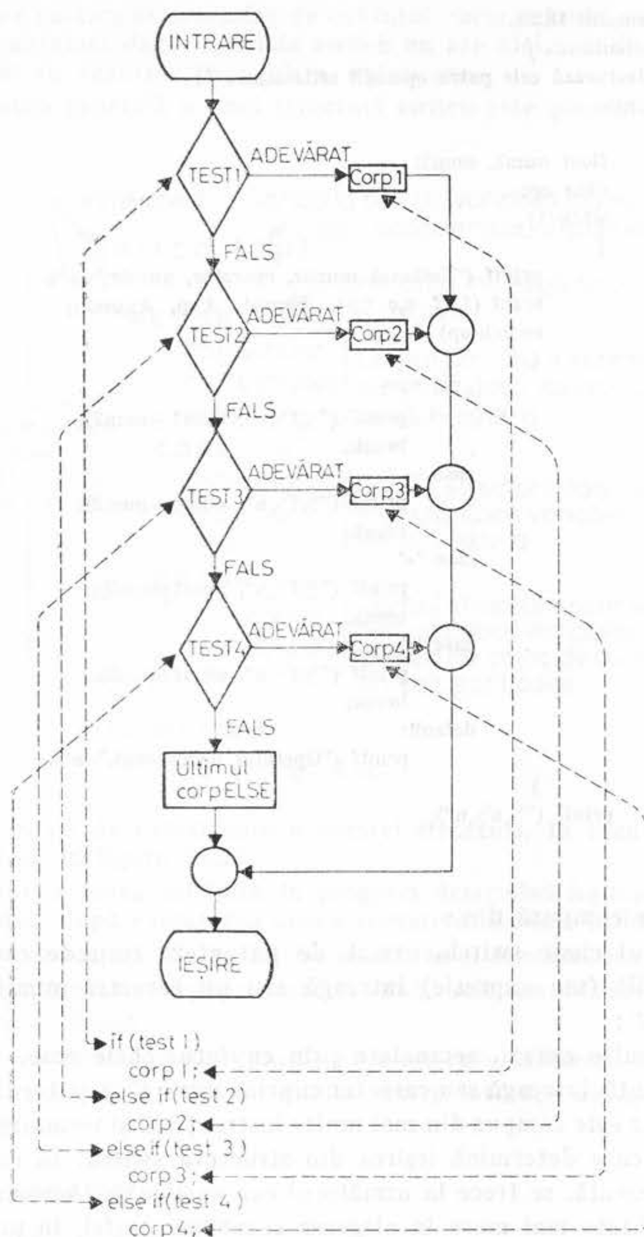


Fig. 18.11.

18.2.6. Structura switch. Instrucțiunile break și continue

Structura **switch** este similară construcției **else-if**, dar are un format mai clar și are mai multă flexibilitate. Ea este analogul structurii „case" din limbajul Pascal. Pentru analiza acestei structuri să rescriem cu ajutorul ei programul de mai sus (calcul 1.c).

Exemplul 18.23.

```
/*calcul2 .c* /
/*efectuează cele patru operaţii aritmetice* /
main()
{
        float num1, num2;
        char op;
        while(1)
        {
                printf ("Tastează număr, operator, număr:\n");
                scanf ("%f %c %f", &num1, &op, &num2);
                switch(op)
                {
                        case '+':
                                printf ("%f\n", num1+num2);
                                break;
                        case '—':
                                printf ("%f\n", num1—num2);
                                break;
                        case '*':
                                printf ("%f\n", num1*num2);
                                break;
                        case '/':
                                printf ("%f\n", num1/num2);
                                break;
                        default:
                                printf ("Operator necunoscut.\n");
                }
        printf ("\n\n");
        }
}
```

Structura este compusă din :

- cuvîntul cheie **switch**, urmat de paranteze rotunde care conţin o variabilă (sau expresie) întreagă sau un caracter numit „**variabilă switch**" ;
- mai multe cazuri, semnalate prin cuvîntul cheie **case**, urmat de o constantă întreagă sau caracter cuprins între (' ') şi terminată cu (:).

Fiecare caz este compus din mai multe instrucţiuni şi terminat cu instrucţiunea **break**, care determină ieşirea din structura **switch**. În cazul în care ea nu este prezentă, se trece la următorul caz particular. Aceasta poate asigura o flexibilitate mai mare în alegerea cazurilor. Astfel, în programul de mai sus putem scrie :

```
. . . . . . . . . . . . . . . . .
        . case '*':
          case 'x':
                printf ("%f\n", num1*num2);
                break ;
. . . . . . . . . . . . . . . . .
```

ceea ce permite alegerea a două simboluri pentru operaţia de înmulţire.

— un caz particular semnalat de cuvîntul cheie **default**, care se va executa automat dacă variabila **switch** nu are nici una din valorile cuprinse în cazurile particulare anterioare.

Componenta generală a unei structuri **switch** este prezentată în figura 18.12.

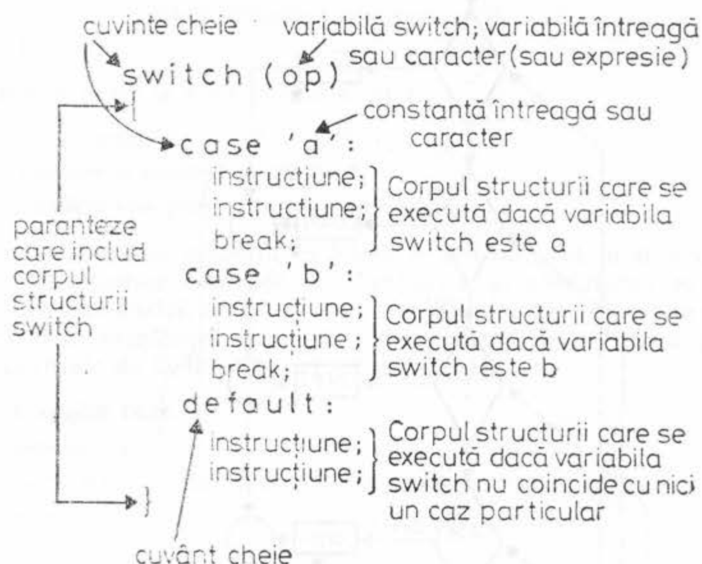


Fig. 18.12.

Schema logică de funcționare a acestei structuri, în cazul exemplului 18.23, este dată în figura 18.13.

Instrucțiunea **break** întîlnită în program determină ieșirea forțată din structura **switch**, după executarea unei alternative. Ea este utilizată în general pentru ieșirea forțată din bucle. La întîlnirea și executarea ei, programul sare la prima instrucțiune din program de după buclă sau de după construcția **switch** care o utilizează.

Să mai considerăm un exemplu de program care utilizează instrucțiunea **break**. Programul decide dacă un număr întreg introdus de operator de la consolă este prim.

Exemplul 18.24.

```
/*prim .c*/
/*stabilește dacă un număr întreg este prim*/
main()
{
        int n;
        int i=1;
        printf ("Introduceți numărul (<32767):");
        scanf ("%d", &n);
        while (++i<n)
                if(n%i==0)
```

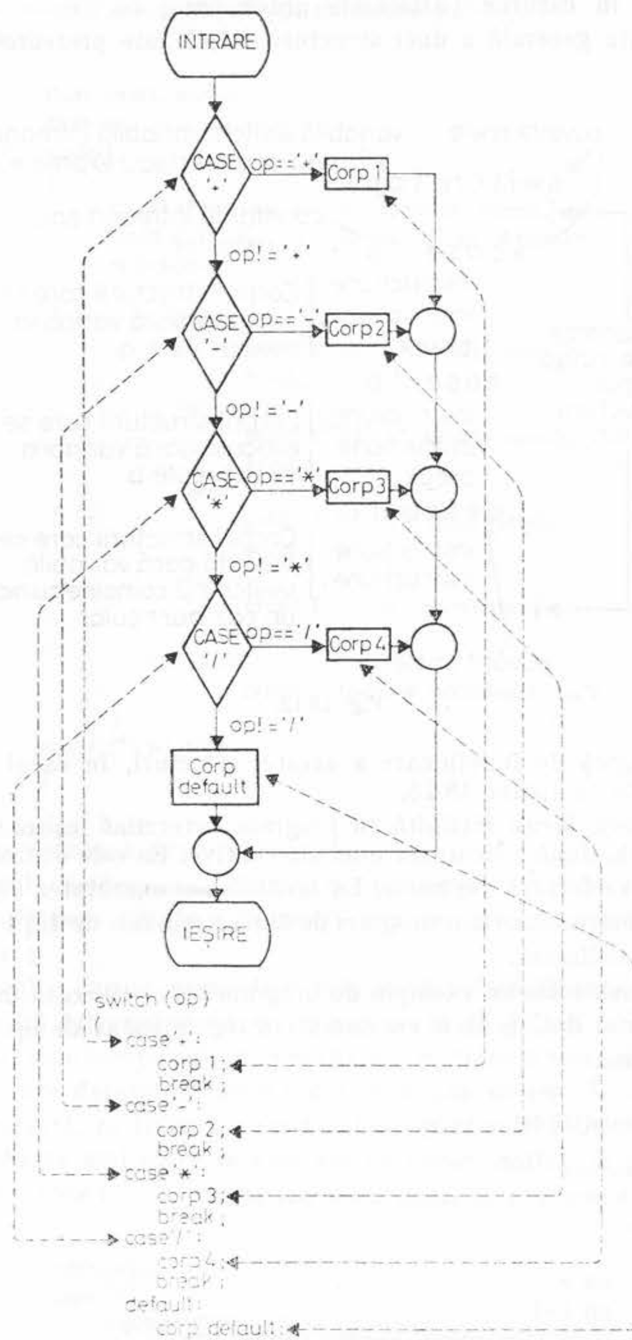

INTRARE
CASE '+'
op=='+'
Corp 1
op!='+'
CASE '-'
op=='-'
Corp 2
op!='-'
CASE '*'
op=='*'
Corp 3
op!='*'
CASE '/'
op=='/'
Corp 4
op!='/'
Corp default
IEȘIRE
switch (op)
case '+':
corp 1;
break;
case '-':
corp 2;
break;
case '*':
corp 3;
break;
case '/':
corp 4;
break;
default:
corp default;

Fig. 18.13.

```
                {
                        printf ("Numărul nu este prim.\n");
                        break;
                }
          if (i==n)
                printf ("Numărul este prim.\n");
     }
```

O execuţie posibilă a programului ar fi

```
C:\> prim
Introduceţi numărul (<32767):17.
Numărul este prim
```

O altă instrucţiune înrudită cu **break** şi utilizată tot în structurile iterative este instrucţiunea **continue**. La întîlnirea şi executarea ei programul revine la începutul buclei, sărind peste instrucţiunile din buclă care o urmează.

Să urmărim următorul exemplu de program care afişează pe display numerele naturale de la 0 la 9 omiţînd numărul 5.

Exemplul 18.25.

```
/*count .c*/
/*afişează numerele naturale 0÷4,  6÷9*/
main()
{
      int i=0;
      while (++i<=9)
      {
            if(i==5)
                  continue;
            printf ("%d\n", i);
      }
}
```

La lansarea în execuţie va apare:

```
C:\>count
     1
     2
     3
     4
     6
     7
     8
     9
```

18.2.7. Operatorul condiţional

O altă construcţie decizională în limbajul C o constituie acest operator condiţional care are următoarea formă generală:

Conditie ? expresie1 : expresie2;

„Condiție ? este o *expresie logică care poate fi adevărată sau falsă.*

„Expresie1" și „expresie2" sunt *valori sau expresii care capătă valori.*

Modul de lucru al acestei structuri este următorul : se evaluează condiția logică („condiție ?"), care, dacă este adevărată, întreaga expresie condițională capătă valoarea „expresiei1" ; dacă condiția este falsă, expresia condițională capătă valoarea „expresie2". Pentru a înțelege mai bine mecanismul, să considerăm următorul exemplu :

```
max = (num1 > num2) ?num1 : num2;
```

în această instrucțiune variabila „max" capătă valoarea „num1" dacă **num1 > num2** și capătă valoarea „num2" dacă **num1 ⩽ num2**. Putem observa că această expresie este echivalentă cu o construcție *if-else*, dar este mult mai compactă :

```
if(num1< num2)
        max=num2;
else
        max=num1
```

18.2.8. Instrucțiunea goto

Instrucțiunea **goto** *permite efectuarea unui salt necondiționat în program la o instrucțiune marcată de o etichetă.* Ea nu poate realiza salt din programul apelant în funcție sau din funcție în programul apelant.

Este o instrucțiune mai puțin folosită, deoarece se arată că *poate fi omisă din orice program.* Altfel spus, nu există cerință de programare care să nu poată evita folosirea acestei instrucțiuni. Să urmărim un exemplu de utilizare a acestei instrucțiuni.

Exemplul 18.26.

```
/*gotoloop.c*/
/*tipărește numerele de la 1 la 99*/
main()
{
        int x=0;
        loop1:                          /*eticheta*/
                x++;
                if (x<100)
                {
                        printf ("\n%d\n", x);
                        goto loop1;     /*instrucțiune de salt
                        ;                necondiționat*/
                }
```

18.2.9. Exerciții

6. Este următoarea structură **switch** corectă ?

```
switch (num)
{
      case 1;
            printf ("Num este 1");
      case 2;
            printf ("Num este 2");
      default:
            printf ("Num nu este nici 1 nici 2.");
}
```

7. Este următoarea structură **switch** corectă ?

```
switch(temp)
{
      case temp<60:
            printf ("Chiar este frig!");
            break;
      case temp<80:
            printf ("Ce vreme incântătoare!");
            break;
      default
            printf ("E prea cald!");
```

8. Scopul operatorului condițional este de a :

a) alege dintre două valori pe cea mai mare ;

b) decide dacă două valori sunt egale ;

c) alege alternativ una din două valori ;

d) alege una din două valori în funcție de o condiție.

9. Dacă valoarea lui num este —42, care este valoarea următoarei expresii condiționale

$$(num < 0)\ ?\ 0 : num*num;$$

10. Scrieți un program care calculează și afișează retribuția unui salariat funcție de grupa de vechime în care se află. Există trei grupe de vechime (1÷3). Retribuția se calculează cu relația :

$$retribuție = salariu + spor$$

Sporul depinde de grupa de vechime și este egal respectiv cu 150, 250, 350. Grupa de vechime și salariul se introduc de operator de la tastatură. Programul va fi pus să lucreze într-o buclă infinită.

11. Scrieți un program care să calculeze și să afișeze rădăcinile reale ale unei ecuații de gradul 2 :

$$ax^2 + bx + c = 0$$

Coeficienții a, b, c, se introduc de la tastatură. Se vor analiza toate cazurile, iar în cazul rădăcinilor complexe se va tipări doar un mesaj.

Se va folosi funcția din biblioteca aritmetică **sqrt()**, care extrage rădăcina pătrată dintr-un număr :

double sqrt (double num)

Programul va lucra într-o buclă infinită.

Observaţie : pentru ca programul să poată fi compilat, în faţa lui **main()** se va scrie instrucţiunea **#include <math.h>**.

12. Scrieţi un program care să returneze un mesaj funcţie de o serie de date introduse de operator de la consolă. Aceste date se referă la un şofer auto relativ la regimul circulaţiei rutiere pe care l-a adoptat :

— a circulat în localităţi sau în afara localităţilor ;

— a circulat cu viteze <60, >60, >80, >120.

Conţinutul mesajului returnat este ales de programator dar este bine să fie corelat cu legislaţia rutieră în vigoare.

13. Se introduc 5 numere întregi de la tastatură. Să se scrie un program care să stabilească care este cel mai mare dintre ele şi să-l afişeze. Se recomandă să se utilizeze operatorul condiţional.

CAPITOLUL 19

FUNCȚII

19.1. GENERALITĂȚI

Funcțiile în limbajul de programare C au același scop ca funcțiile sau procedurile în Pascal. Principala justificare a introducerii funcțiilor constă în *evitarea scrierii repetate a aceluiași cod într-un program sau în programe diferite*, ceea ce conduce evident la *economie de memorie* (fig. 19.1). Totodată, prin folosirea funcțiilor se ușurează alcătuirea și înțelegerea programelor și se poate realiza și o modularizare a lor prin posibilitatea de scriere și testare a funcțiilor separat.

În limbajul Pascal, procedurile și funcțiile sînt două construcții diferite. O funcție în acest limbaj returnează o valoare (prin numele funcției), în timp ce o procedură returnează eventual o dată printr-un argument al procedurii. În limbajul C se operează numai cu funcții care combină cele două construcții de mai sus. Astfel, o funcție în C poate returna o valoare (prin numele funcției), dar poate returna și o dată printr-un argument.

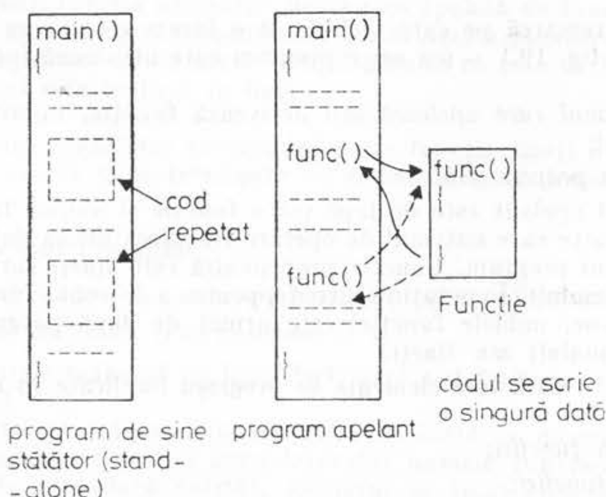


Fig. 19.1

19.2. STRUCTURA PROGRAMELOR CARE UTILIZEAZĂ FUNCȚII

Pentru a analiza problematica utilizării funcțiilor în limbajul C, să începem cu un exemplu simplu :

Exemplul 19.1.

```
/*chenar .c*/
/*realizează un chenar de '*' în jurul unui nume*/
void line (void);
main()
{
        line();
        printf ("* POPESCU ION *\n");
        line();
}
/*line() — este numele funcţiei*/
/*trasează o linie de '*' pe diplay*/
void line (void)
{
        int j ;
        for (j=1; j<=15; j++)
                printf ("*");
        printf ("\n");
}
```

În urma executării acestui program, numele Popescu Ion va fi încadrat cu '*' fig. 19.2.

```
***************
* POPESCU ION *
***************
```

Fig. 19.2.

O primă remarcă pe care trebuie să o facem este aceea că programul de mai sus – fig. 19.1 – (ca orice program care utilizează funcții) are **două componente:**

- *programul care apelează* sau activează funcția, numit și *program apelant*;
- *funcția propriu-zisă.*

Programul apelant este de fapt tot o funcție și anume funcția privilegiată **main()** către care sistemul de operare transferă prima dată controlul la executarea unui program. Funcția propriu-zisă este **line()** care este apelată de două ori de **main()**. În notațiile curente, pentru a deosebi o funcție de oricare altă instrucțiune, numele funcției este urmat de două paranteze rotunde. De exemplu, **main()** sau **line()**.

În general există **trei elemente de program** implicate în utilizarea unei funcții :

1. *definiția funcției;*
2. *apelul funcției;*
3. *prototipul funcției.*

19.2.1. Definiția funcțiilor

Definiția funcției (fig. 19.3) începe cu o linie care include printre altele numele funcției.

void line (**void**) — **fără terminator(;)**

Această linie formează ceea ce numim **declaratorul funcției** care nu este o instrucțiune, ci o *informare pentru compilator*.

```
void line (void) ◄— declaratorul funcției
{
   int j;
   for (j=1; j<=15; j++)          corpul
        printf("*");               funcției
   printf("\n");
}
```

Fig. 19.3.

Primul **void** din linie, specifică faptul că funcția **line()** nu returnează nimic în programul apelant, iar al doilea **void** specifică faptul că funcția **line()** nu are parametri (programul apelant nu-i transferă nici un parametru la apelarea ei). Tipul **void** de variabilă a fost introdus de ANSI în 1983, pentru a inlătura ambiguitățile de pînă atunci în cazul în care funcțiile nu aveau parametri sau nu returnau nici o valoare în programul apelant. De exemplu, pînă atunci o funcție care nu returna nici o valoare în programul apelant se considera implicit că returnează o valoare intreagă.

După declarator urmează corpul funcției care formează un bloc cuprins între paranteze acolade.

19.2.2. Apelul funcțiilor

Ca și în cazul funcțiilor din biblioteca C folosite pînă acum, **printf()**, **scanf()**, **getche()**, funcția utilizator **line()** este apelată de funcția **main()** prin numele ei, urmat de două paranteze rotunde. Aceste paranteze sânt necesare deoarece cu ajutorul lor se arată compilatorului că este vorba de o funcție. Apelul funcției este realizat în linia :

line();

și va determina transferul controlului către funcția **line()** definită mai sus, care se va executa și la terminare va retransmite controlul în programul apelant.

19.2.3. Prototipul funcțiilor

Prototipul funcției îl constituie linia de program aflată înainte de **main()**:

```
void line (void); /* cu terminator (;) */
```

Această linie seamănă cu linia declarator a definiției, cu diferența că se termină cu (;).

Cu ajutorul prototipului funcția este declarată la începutul programului. Prin aceasta se specifică compilatorului numele funcției, tipul datelor returnate de funcție (dacă există), numărul și tipul argumentelor funcției (dacă există). Prin scrierea prototipului înainte de **main()**, funcția devine vizibilă din toate funcțiile programului.

Este obligatoriu ca tipul datelor utilizate în prototip să corespundă cu cele existente în declarator ; în caz contrar compilatorul va semnala eroare.

În concluzie putem sintetiza cele spuse mai sus astfel :

— *prototipul declară funcția* ;

— *apelul funcției execută funcția* ;

— *declaratorul funcției* (din definiție) *specifică numele funcției și precizează tipul argumentelor și tipul valorii returnate.*

19.3. VARIABILE LOCALE

O particularitate foarte importantă care trebuie însușită ori de cîte ori se lucrează cu funcții în limbajul C, o constituie faptul că *o variabilă definită într-o funcție este recunoscută numai în acea funcție* fiind invizibilă în alte funcții sau în programul apelant. Astfel, variabila j definită și utilizată în **line()**, este văzută numai în această funcție fiind invizibilă din **main()**. Dacă am fi declarat o variabilă în **main()** ea ar fi fost recunoscută numai aici și ar fi fost complet necunoscută în **line()**.

Aceste variabile care sînt vizibile numai în funcțiile în care au fost declarate și sînt invizibile în afara acestor funcții, poartă denumirea de **variabile locale sau automate.** Ele au un timp de viață limitat, fiind create automat la apelul funcției și distruse la terminarea funcției.

19.4. FUNCȚII CARE RETURNEAZĂ O VALOARE

Funcții care returnează o valoare, dar nu necesită parametri am întâlnit și până acum. Am utilizat în exemple funcția **getche()**, care returnează în programul apelant un caracter (cel tastat de operator). Să urmărim un alt exemplu în care introducem o astfel de funcție creată însă de operator. Programul primește două valori de timp în ore și minute, convertește aceste valori în minute cu ajutorul funcției și calculează diferența în minute între aceste două valori.

Exemplul 19.2.

```
/*dif_time.c*/
/*calculează diferența între două valori de timp*/
int getmin (void) ; /*prototipul*/
main()
{
        int min1, min2;
        while (1) /*bucla infinită*/
        {
                printf ("Tipăriți primul timp (forma 5:25):\");
                min1=getmin();
                printf ("Tipărește al doilea timp (forma 5:25):\");
                min2=getmin();
                printf ("Diferența este %d minute\n", min2—min1");
        }
}
/*funcția getmin*/
/*cere timpul în format ore: minute*/
/*returnează timpul în minute*/
int getmin (void) /*declarator*/
```

```
{
    int ore, min, total;
    scanf ("%d: %d", &ore, &min); /*utilizatorul introduce timpul*/
    total=ore*60+min ; /*converteşte timpul în minute*/
    return (total);
}
```

Funcţia **getmin()** este apelată de două ori în **main()**. Funcţia returnează la fiecare apel câte o valoare tip întreg atribuită pe rând variabilelor min1 şi min2.

O execuţie a programului poate fi :

```
C:\> dif_time.
Tipăriţi primul timp (forma 5:25):3:22
Tipăriţi al doilea timp (forma 5:25):4:15
Diferenţa este 53 de minute
Tipăriţi primul timp (forma 5:25)
. . . . . . . . . . . . . . . . .
```

În funcţiile care returnează valori în programul apelant, valoarea returnată este plasată între paranteze în instrucţiunea **return**. Această instrucţiune are două acţiuni :
— transferă imediat controlul de la funcţie în programul apelant ;
— ceea ce este între parantezele ei este returnat ca valoare în programul apelant.

Tipul variabilei returnate este specificat atât în prototipul, cât şi în declaratorul funcţiei ; în acest caz se returnează o valoare de tip întreg.

Instrucţiunea **return** nu este necesar să fie la sfârşitul funcţiei ; ea poate fi oriunde în corpul funcţiei. Dacă cuvântul cheie **return** apare singur (nu este urmat de paranteze), înseamnă că nu se returnează nici o valoare în programul apelant. Astfel, prin instrucţiunea **return** se returnează cel mult o valoare în programul apelant.

Cu ajutorul exemplului de mai sus, punem în evidenţă noi posibilităţi ale funcţiei **scanf()**. Datele pot fi introduse de la tastatură folosind şi separatorul(:) în loc de *spaţiu, tab* sau *newline*, dacă acest lucru este specificat în cadrul funcţiei (se pun (:) între două specificaţii de format).

```
scanf ("%d : %d", &ore, &min);
```

19.5. FUNCŢII CU PARAMETRI (ARGUMENTE)

Argumentele sau parametrii permit transferul de informaţii din programul apelant în funcţie. Până acum am folosit funcţii cu parametri, şi anume **printf()** şi **scanf()**. La aceste funcţii, parametrii erau şirul de caractere şi variabilele dintre paranteze. Să dăm un exemplu de program care utilizează o *funcţie căreia i se transferă un singur argument*. Programul primeşte un număr introdus de operator şi trasează pe ecranul display-ului o linie cu un număr corespunzător (*)

Exemplul 19.3.

```
/*star .c*/
/*trasează o linie de (*)*/
void bar (int); /*prototip*/
main()
```

```
{
        int num;
        while (1)
        {
                        printf ("Introduceţi numărul:");
                        scanf ("%d", &num);
                        bar (num);
        }
}
/*bar()*/
/*trasează o linie de (*)*/
void bar (int val)          /*declarator*/
{
        int j;
        for (j=1; j<=val; j++)
                printf ("*");
        printf ("\n");
        return;
}
```

În urma execuţiei programului rezultă :

```
C:\>star
Introduceţi numărul: 15
***************
Introduceţi numărul: 7
*******
Introduceţi numărul: . . . . . . . .
```

Programul lucrează într-o buclă infinită, ieşirea din el făcîndu-se cu ajutorul combinaţiei de taste [CTRL] [C].

Funcţia bar() utilizată în acest program este o funcţie care necesită argumente, dar nu returnează nimic în programul apelant (este de tip **void**). De aceea, instrucţiunea **return** care apare la sfîrşitul corpului funcţiei nu este urmată de paranteze. În astfel de cazuri în care **return** apare singură la sfârşitul corpului funcţiei, ea poate lipsi din programul funcţiei, terminarea fiind sesizată de paranteza (}).

Parametrul pe care dorim să-l transmitem funcţiei apare sub formă de constantă sau nume de variabilă între parantezele care urmează funcţia

bar(num);

Tipul variabilei sau constantei care se transferă trebuie specificat atât în prototipul funcţiei, cât şi în definiţia sa şi trebuie să coincidă cu cel care se transferă în realitate, altfel se vor semnala erori în diferite etape de evoluţie a programului. Între prototip şi declarator apare în acest caz o diferenţă :
— prototipul trebuie să conţină în mod obligatoriu tipul variabilei care se transmite şi poate conţine în mod opţional şi numele ei;
— declaratorul va conţine obligatoriu numele variabilei transferate, tipul ei putînd fi definit tot aici sau în corpul funcţiei.

Ca atare s-a convenit să se adopte următoarea **regulă de definire a funcțiilor cu parametri:**

— *prototipul va conține numai tipul variabilei care se transferă;*

— *declaratorul va conține atât tipul, cât și numele variabilei care se transferă* (fig. 19.4).

```
void bar(int);          /* prototip */

void bar (int val)  /* declarator */
{
  int j;
  for(j=1;j<=val;j++)
      printf("*");                  corpul
  printf("\n");                     funcției
}
```

Fig. 19.4.

Să mai observăm, că numele „val" al parametrului din declaratorul funcției, diferă de numele parametrului funcției din programul apelant „num". Această nepotrivire de denumiri este acceptată de limbajul C (precum și de orice limbaj de programare de nivel înalt). Argumentul funcției din programul apelant poartă denumirea de *argument sau parametru actual*, în timp ce argumentul ei din declarator se numește *argument sau parametru formal*. Indiferent de numele acestor parametri (tipurile lor trebuie să coincidă), la apelul funcției, valoarea parametrului actual este transferată automat parametrului formal. Cu această ocazie se face o copie a acestei valori din spațiul de memorie rezervat programului apelant într-un alt spațiu de memorie aferent funcției (fig. 19.5).

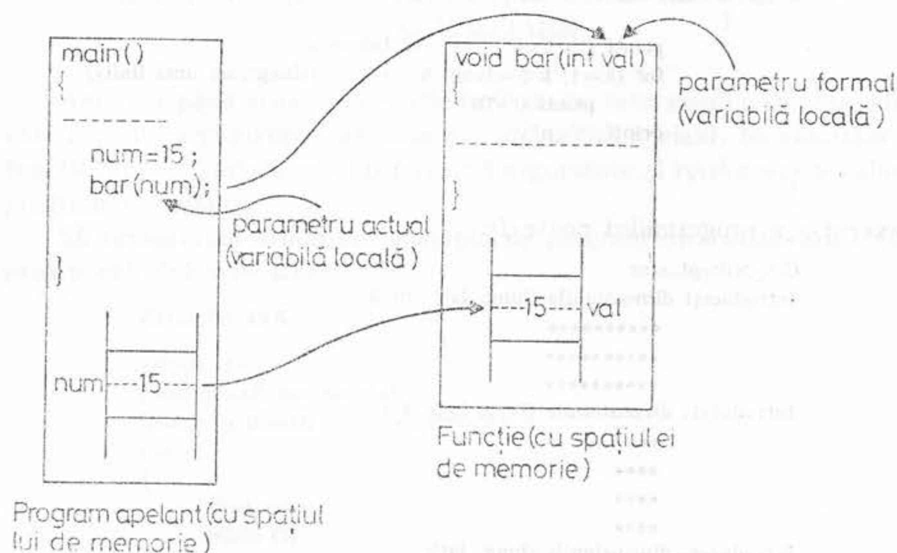


Fig. 19.5.

Această observație este important să fie reținută (avînd în vedere și înțelegerea unor mecanisme din capitolele următoare): spațiul de memorie în care lucrează programul apelant diferă de spațiul de memorie în care lucrează funcția, iar transferul parametrilor înseamnă realizarea de copii cu valorile acestora în cele două spații.

19.6. TRANSFERUL PARAMETRILOR MULTIPLI

O funcție poate primi din programul apelant mai mult de un parametru. Să dăm un exemplu de program în care utilizăm o funcție căreia i se transferă doi parametri. Programul trasează dreptunghiuri formate din (*), de dimensiuni date.

Exemplul 19.4.

```
/*drept_star .c*/
/*trasează dreptunghiuri cu ajutorul (*)*/
void drept (int, int); /*prototip*/
main()
{
        int x, y;
        while (1)
        {
                printf ("Introduceți dimensiunile (lung, lat):");
                scanf ("%d %d" &x, &y);
                drept (x, y)
        }
}
/*funcția drept() trasează pe display*/
/*dreptunghiuri umplute cu (*)*/
void drept (int lung, int lat)      /*declarator*/
{
int j, k;
        for (j=1; j<=lat; j++)      /*numărul de linii*/
        {
                printf ("\t\t") ;       /*2 tab-uri*/
                for (k=1; k<=lung; k++)      /*lungimea unei linii*/
                        printf ("*");
                printf ("\n");
        }
}
```

O execuție a programului poate fi:

```
C:\>drept_star
Introduceți dimensiunile (lung, lat): 10 3
                **********
                **********
                **********
Introduceți dimensiunile (lung, lat): 4 4
                ****
                ****
                ****
                ****
Introduceți dimensiunile (lung, lat):
                . . . . . . .
```

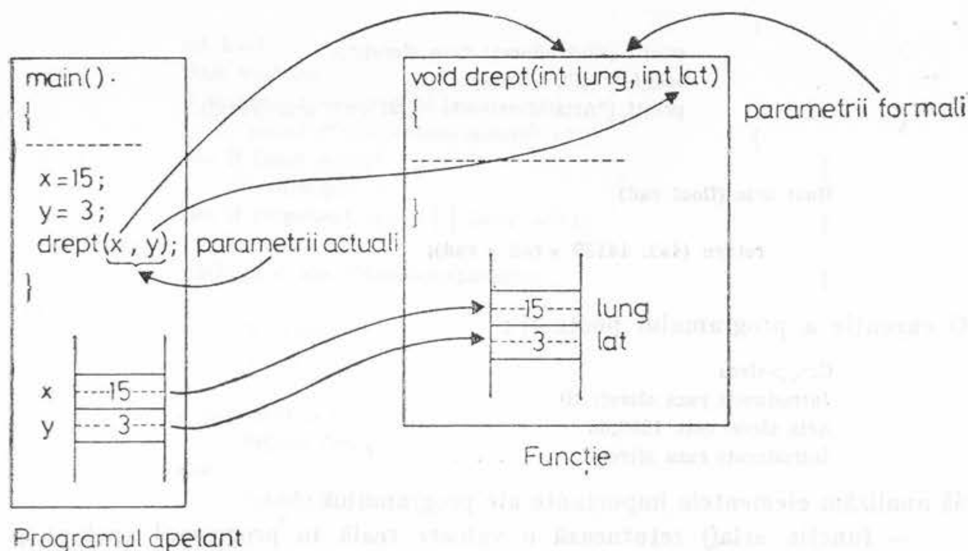


Fig. 19.6.

Se iese din program cu combinația de taste [Ctrl] [C].

Procesul de transfer a două argumente este similar cu cel de transfer al unui singur argument. Valoarea primului argument actual al funcției din programul apelant este atribuită primului argument formal al funcției din definiție, iar valoarea celui de-al doilea argument actual este atribuită celui de-al doilea parametru formal al funcției (fig. 19.6).

În cazul funcțiilor cu mai mult de două argumente, regula de transferare a acestora va fi aceeași.

19.7. FUNCȚII CU PARAMETRI CARE RETURNEAZĂ O VALOARE

Am avut până acum exemple de funcții care returnează o valoare și funcții care necesită argumente transmise din programul apelant. Să analizăm acum funcții care au ambele calități (acceptă argumente și returnează o valoare în programul apelant).

Să considerăm următorul exemplu de program care utilizează o funcție pentru calculul ariei sferei.

Exemplul 19.5.

```
/*sfera .c*/
/*calculează aria sferei*/
float aria (float);          /*prototip*/
main()
{
        float raza;
        while (1)
```

```
        {
            printf ("Introduceţi raza sferei:");
            scanf ("%f", &raza);
            printf ("Aria sferei este %.2f\n", aria (raza));
        }
}
float aria (float rad)
{
    return (4*3. 14159 * rad * rad);
}
```

O execuţie a programului poate fi :

```
C:\>sfera
Introduceţi raza sferei: 10
Aria sferei este 1256.64
Introduceţi raza sferei: . . . . . .
```

Să analizăm elementele importante ale programului :

- funcţia **aria()** returnează o valoare reală în programul apelant şi primeşte de la acesta un parametru real, calitate specificată corect atît în prototipul funcţiei, cît şi în declarator;
- argumentul actual al funcţiei este „raza", iar cel formal „rad", care sînt declarate corect de tip **float** în programul apelant, respectiv în definiţia funcţiei.

Ca atare, programul va lucra corect, fără erori, atat în faza de compilare cât şi în faza de execuţie.

Vom considera în continuare un exemplu de program care utilizează o *funcţie cu doi parametri care returnează o valoare reală.* Programul efectuează ridicarea la o putere întreagă a unui număr real.

Exemplul 19.6.

```
/*putere .c*/
/*ridică un număr real la o putere întreagă*/
#include <math.h>
float putere (int, float)    /*prototip*/
main()
{
    int n;
    float x ;
    while(1)
    {
        printf ("Introduceţi exponentul (int) şi baza (float):");
        scanf ("%d %f", &n, &x);
        printf ("Rezultatul este: %.3f", putere (n, x));
    }
}
/*funcţia putere()*/
/*returnează valoarea obţinută în urma ridicării la o putere întreagă*/
/*a unui număr real pozitiv*/
float putere (int exponent, float baza)
{
```

```
int k=1;
float y=baza;
if (exponent ==0 && baza ==0)
        printf ("\n Nedeterminare\n");
else if (baza ==0)
        return (0);
else if (exponent ==0 || baza ==1)
        return (1);
while (k < abs ((double)exponent))
{
        y*=baza;
        k++;
}
if exponent > 0
        return (y);
else
        return (1/y);
}
```

Programul lansat în execuţie cere operatorului să introducă în ordine exponentul (valoare întreagă) şi baza (valoare reală), după care se va afişa rezultatul operaţiei de ridicare la putere ţinând cont de toate cazurile posibile. Un exemplu de execuţie a programului ar putea fi :

```
C:\>putere
Introduceţi exponentul (int) şi baza (float): 3 2.5
Rezultatul este 15.625
Introduceţi exponentul (int) şi baza (float):  . . . .
```

La prima execuţie s-a afişat rezultatul (cu trei zecimale exacte) operaţiei $(2.5)^3$.

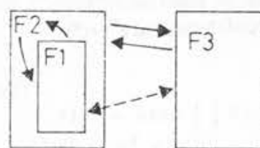
19.8. UTILIZAREA MAI MULTOR FUNCŢII ÎNTR-UN PROGRAM

Într-un program scris în limbajul de programare C *se pot utiliza mai multe funcţii care se pot apela unele pe altele.* Din acest punct de vedere, există o deosebire majoră între limbajele C şi Pascal. În Pascal o funcţie (sau o procedură), numită de exemplu F1, poate fi definită în interiorul altei funcţii numită de exemplu F2. În acest caz F1 va fi invizibilă pentru o altă funcţie F3, care nu se află în F2 (fig. 19.7, *a*). *În limbajul C nu se pot defini funcţii în interiorul altor funcţii.* Toate funcţiile definite în acest limbaj sînt vizibile între ele şi au acelaşi statut, inclusiv funcţia **main()** (fig. 19.7, *b*).

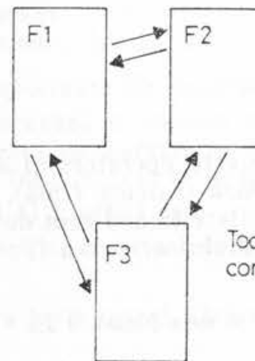
Această particularitate a limbajului C nu-i reduce flexibilitatea şi în plus creează anumite avantaje în programare. Să urmărim un exemplu de program care utilizează trei funcţii în afară de **main()** şi care calculează suma pătratelor a doi întregi introduşi de utilizator de la tastatură.

Exemplul 19.7.

```
/*multifunc.c*/
/*exemplu de utilizare a mai multor funcţii*/
int sumsqr (int, int); /*prototipurile funcţiilor*/
```



COMUNICARE ÎN PASCAL



COMUNICARE ÎN C

Fig. 19.7.

```
int sqr (int);
int sum (int, int);
main()
{
        int num1, num2;
        printf ("Introduceţi două numere întregi:");
        scanf ("%d %d", &num1, &num2);
        printf ("Suma pătratelor este : %d\n", sumsqr(num1, num2));
}
/*funcţia sumsqr()*/
/*returnează suma pătratelor a două numere*/
int sumsqr (int j, int k)     /*declarator*/
{
        return sum(sqr(j), sqr(k));
}
/*funcţia sqr()*/
/*returnează pătratul unui număr*/
int sqr (int z) /*declarator*/
{
        return (z*z);
}
```

```
/*funcţia sum()*/
/*returnează suma a două numere*/
int sum(int x, int y) /*declarator*/
{
      return (x+y);
}
```

În urma execuţiei programului poate rezulta :

```
C:\>multifunc
Introduceţi două numere întregi : 3 5
Suma patratelor este 34
```

Să facem observaţia că în Pascal pentru ca acest program să funcţioneze ar fi trebuit să plasăm funcţiile **sum()** şi **sqr()** în interiorul funcţiei **sumsqr()**. În limbajul C, *toate funcţiile sînt declarate la fel şi sunt vizibile una din alta.* De exemplu, programul principal poate apela direct funcţiile **sum()** şi **sqr()** dacă este nevoie. În plus funcţiile pot fi trecute în orice ordine în listingul programului. Să mai notăm de asemenea că funcţia **main()** nu este necesar să fie prima în program, aceasta este însă uzual.

19.9. STANDARDUL ANSI AL LIMBAJULUI C FAŢĂ DE VERSIUNEA „KERNIGHAN ŞI RITCHIE" INIŢIALĂ

Între cele două implementări ale limbajului C există o serie de **diferenţe notabile**.

1. *În versiunea clasică realizată de Kernighan şi Ritchie, funcţiile nu aveau prototip.* Această lipsă a prototipului putea genera frecvent eroarea de a apela o funcţie utilizînd tipuri greşite de date pentru argumente (de exemplu **int** în loc de long, etc.). Dacă apelul funcţiei şi definiţia funcţiei erau în fişiere diferite, la execuţie programul eşua fiind greu de depistat cauza şi greu de depanat. *Utilizînd prototipul, compilatorul este informat ce tip de date necesită funcţia* şi va semnala eroare la compilare, în caz de nepotrivire a argumentelor.

Să dăm un exemplu de program simplu care utilizează o funcţie şi să-l scriem conform standardului **ANSI** şi în versiune clasică. Funcţia preia o valoare întreagă din programul apelant şi o tipăreşte.

Exemplul 19.8.

```
/*proto.c*/
  /*program scris conform standardului ANSI*/
  void func (int);   /*prototip*/
main()
{
      int num=1234;
      func (num);
}
/*funcţia func()*/
/*afişează la display valoarea întreagă preluată din main()*/
void func (int val)   /*declarator*/
{
      printf ("Argumentul are valoarea: %d\n", val);
}
```

În acest program nu sunt surprize ; textul corespunde cu cele spuse mai sus.

Exemplul 19.9.

```
/*noproto.c*/
/*program scris în versiunea clasică*/
main()
{
        int num=1234;
        func (num);
}
/*funcţia func()*/
func (val)     /*declarator*/
int val;
{
        printf ("Argumentul are valoarea: %d\n" val);
}
```

2. *În versiunea clasică, declaratorul funcţiei şi declaraţia argumentului sunt două linii de program distincte.*

3. *În versiunea clasică, în cazul în care tipul datei returnate de funcţie nu este precizat, se consideră implicit că este de tip întreg.* În acest caz pot apare confuzii în cazul în care funcţia nu returnează nimic (cum apare în exemplul nostru). Funcţia care nu returnează nimic este considerată tot de tip **întreg**. Ca atare, standardul **ANSI** a introdus tipul de dată **void** pentru a departaja cele două situaţii.

Implementările moderne ale limbajului C (Microsoft C, Turbo C, Borland C, Microware etc.) *sunt realizate conform standardului ANSI.* Unele dintre aceste versiuni includ însă şi versiunea clasică „**K-R**". Aceasta înseamnă că programe scrise conform acestei versiuni (exemplul 19.9) pot fi compilate şi rulate fără să se semnaleze erori pe echipamente dotate cu compilatoare moderne de C. Acesta este un principiu care trebuie respectat la trecerea de la o versiune mai veche la una nouă a unui limbaj de programare. Altfel, tot efortul de programare făcut pînă în acel moment, ar trebui reconsiderat, ceea ce ar crea o anumită aversiune faţă de noua versiune şi chiar faţă de limbajul respectiv.

19.10. VARIABILE EXTERNE

Variabilele utilizate până în prezent în programe erau vizibile numai din funcţiile în care erau declarate şi care le utilizau. Datorită acestei particularităţi ele se numesc **variabile locale** sau **variabile automate.** Există situaţii însă în care dorim ca o variabilă să fie văzută din toate funcţiile programului şi nu numai din cea în care a fost declarată. În acest caz trebuie să utilizăm **variabile externe** sau **variabile globale.**

În programul următor vom arăta cum se declară o variabilă externă şi care este vizibilitatea ei în program. Programul cere introducerea unui număr de la tastatură şi testează cu ajutorul a două funcţii paritatea şi semnul numărului.

Exemplul 19.10.

```
/*extern.c*/
/*utilizează variabile externe*/
void impar (void);
void negativ (void);
int num;
main()
{
        printf ("Introduceţi numărul:");
        scanf ("%d", &num);
        impar();
        negativ();
}
/*funcţia impar()*/
/*determina paritatea numărului*/
void impar (void)
{
        if (num %2)
                printf ("Numărul este impar.\n");
        else
                printf ("Numărul este par.\n");
}
/*funcţia negativ()*/
/*determină semnul numărului*/
void negativ (void)
{
        if (num < 0)
                printf ("Numărul este negativ.\n");
        else
                printf ("Numărul este pozitiv.\n");
}
```

În acest program *num* este *variabilă externă* şi ea este văzută atât din funcţia **main()**, cât şi din celelalte două funcţii **impar()** şi **negativ()**. Pentru a crea acest statut global al variabilei, este necesar ca ea să fie declarată în afara tuturor funcţiilor (şi în afara funcţiei **main()**). Ca atare, declararea unei *variabile externe* se face înaintea cuvântului cheie **main()**.

Execuţia programului ar putea fi :

```
C:\>extern
Introduceţi numărul: 25
Numărul este impar.
Numărul este pozitiv.
```

Urmărind acest program, ar putea apare la prima vedere impresia că s-ar realiza o simplificare a programelor în limbajul C dacă toate variabilele ar fi declarate „*extern*". Dar în acest caz există două mari inconveniente :

— se realizează o utilizare ineficientă a memoriei;

— apare posibilitatea de alterare accidentală a acestor variabile externe.

Ca atare, se recomandă să utilizăm *variabile externe* numai în cazul în care raţiuni deosebite o cer. În rest se vor utiliza numai variabile locale.

19.11. DIRECTIVE CĂTRE PREPROCESOR

Suntem în momentul în care putem explora o topică din limbajul C, care la prima vedere nu pare a avea legătură directă cu funcțiile: este vorba de utilizarea **directivelor către preprocesor.** Acestea formează ceea ce poate fi considerat un *limbaj în interiorul limbajului C.* Aceste facilități nu există în alte limbaje de programare de nivel înalt, ele fiind similare cu cele existente în limbajele de asamblare. Pentru a înțelege ce sînt aceste directive către preprocesor, să recapitulăm ce face un compilator. Când scriem în program o instrucțiune, de exemplu:

```
num=75;
```

cerem compilatorului să translateze acest cod în limbaj mașină care poate fi executat de microprocesor. Astfel, cea mai mare parte din listingul programului constă în instrucțiuni destinate microprocesorului (sau procesorului). Directivele către preprocesor sunt *instrucțiuni destinate compilatorului* (preprocesorului) care *trebuie să execute ceva înainte de a începe compilarea.* Cele mai utilizate directive către preprocesor sunt **#define** și **#include.**

În general, directivele către preprocesor se recunosc după simbolul (#) care apare în fața lor. Ele pot fi puse oriunde în program, dar cel mai frecvent apar la începutul lui, înainte de **main()** sau înainte de începutul unei funcții particulare.

19.11.1. Directiva #define

Cea mai simplă utilizare a directivei **#define** este de a atribui un nume unor constante. Să revenim la programul din exemplul 19.5 și să-l modificăm în acest sens.

Exemplul 19.11.

```
/*sfera.c*/
/*calculează aria sferei*/
#define PI 3.14159
float aria (float);          /*prototip*/
main()
{
        float raza;
        while(1)
        {
                printf ("Introduceți raza sferei:");
                scanf ("%f", &raza);
                printf ("Aria sferei este %.2f\n", aria(raza));
        }
}
float aria (float rad)
{
        return (4 * PI * rad * rad);
}
```

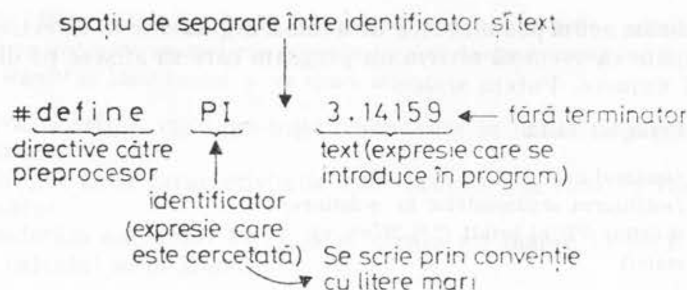


Fig. 19.8.

Compilatorul (preprocesorul) caută întâi în toate liniile de program simbolul #. Cînd găseşte directiva **#define**, el baleiază din nou tot programul şi înlocuieşte peste tot pe PI cu 3.14159, după care va începe compilarea programului. Structura directivei **#define** este arătată în figura 19.8.

O primă observaţie ne arată că utilizarea directivei **#define** prezintă o serie de avantaje:

1. *Programul devine mai uşor de citit* (mai ales în cazurile în care se înlocuieşte un cod cu numele acţiunii lui).
2. *Se poate modifica uşor valoarea constantei* (textului) *atribuit identificatorului, fără a interveni în program.*

Este evident că la prima vedere acelaşi efect s-ar fi obţinut dacă în locul directivei **#define** s-ar fi utilizat o variabilă „pi" căreia i s-ar fi atribuit în program valoarea 3.14159. Se preferă însă varianta cu directiva **#define**, deoarece se generează un cod mai rapid şi mai compact (utilizînd o constantă în locul unei variabile) şi se obţine un program mai bine protejat (o variabilă poate fi alterată involuntar) şi mai uşor de citit.

19.11.2. Macroinstrucţiuni

Directiva **#define** este mult mai puternică decît apare la prima vedere. Forţa ei constă în posibilitatea de a utiliza argumente. Înainte de a pune în evidenţă această facilitate, să considerăm încă un exemplu de utilizare a ei pentru a face tranziţia mai clară. În acest exemplu vom vedea că directiva poate fi utilizată nu numai pentru substituirea constantelor, ci şi pentru substituirea oricărei alte expresii. Să presupunem că programul trebuie să scrie mesajul „Eroare" în diferite situaţii. Utilizînd directiva **#define** putem scrie:

```
#define ERROR printf ("\nEroare\n");
```

Dacă în program există secvenţa:

```
if (val > 200)
ERROR
```

înainte de compilare, se va face înlocuirea:

```
if (val > 200)
        printf ("\nEroare\n");
```

de către preprocesor.

Observăm deci, şi aceasta este foarte important de reţinut, că un *identificator definit prin* **#define** *poate fi substituit cu o instrucţiune.*

Să analizăm acum posibilitatea de a folosi argumente în directiva **#define**. Să presupunem că vrem să scriem un program care să afișeze pe display valorile a două numere. Putem scrie :

Exemplul 19.12.

```
/*macro1.c*/
/*utilizarea argumentelor în #define*/
#define PR(n) printf ("%.2f/n", n);
main()
{
        float num1=12.5;
        float num2;
        num2+num1/3;
        PR(num1)
        PR(num2)
}
```

La execuția programului pe display va apare :

```
C:\>macro1
12.5
4.16
```

În acest program, compilatorul (preprocesorul) înainte de compilare, baleind programul și întâlnind expresia PR(n) o va înlocui cu instrucțiunea :

```
printf ("%.2f\n",n);
```

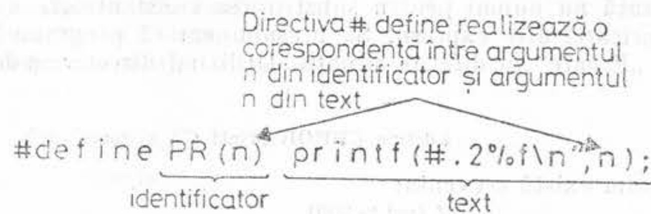
Variabila „*n*", existentă în identificatorul PR(n), poartă denumirea de **argument** și el este pus în corespondență cu argumentul „*n*" din instrucțiunea **printf()** care formează textul. În aceste condiții instrucțiunea PR(num1) din program va determina substituirea variabilei n cu num1, adică va fi înlocuită cu :

```
printf ("%.2f\n", num1);
```

De asemenea, PR(num2) va fi echivalentă cu :

```
printf ("%.2f\n", num2);
```

În figura 19.9 este arătat schematic cum are loc acest proces.



PR(num1) ← Acest identificator din program

va fi expandat în următorul text

printf("%.2f\n",num1); ←

argumentul se inserează nealterat

Fig. 19.9.

Observaţie importantă: în expresia directivei **#define** nu trebuie lăsat nici un spaţiu între caracterele identificatorului, deoarece acesta va fi interpretat ca sfârşit de identificator şi va apare eroare.

O directivă **#define**, care foloseşte argumente în modul utilizat mai sus se numeşte **macro**.

Macrourile pot avea caracteristicile unor funcţii, aşa cum va rezulta din exemplul următor.

Să reconsiderăm exemplul 19.11 şi să creăm un **macro** în locul funcţiei **aria()** pentru calculul ariei sferei.

Exemplul 19.13

```
/*sfera2.c*/
/*calculează aria sferei utilizând macrouri*/
#define PI 3.14159
#define ARIA(x) (4*PI*x*x)    /*macro*/
main()
{
    float raza;
    printf ("Introduceţi aria sferei:");
    scanf (" %f", &raza);
    printf ("Aria sferei este %.2f\n", ARIA(raza));
}
```

În acest program, înainte de compilare preprocesorul va pune în locul identificatorului ARIA(raza), textul (4*3.14159*raza*raza). Să notăm că aici am utilizat un identificator (PI) în interiorul altui identificator ARIA(x).

În condiţiile în care un **macro** sau o **funcţie** pot avea acelaşi efect asupra unui program, se pune întrebarea legitimă când este bine să utilizăm un **macro** şi cînd este bine să utilizăm o **funcţie**? Pentru a putea da un răspuns la această întrebare să analizăm mai atent cum lucrează cele două mecanisme.

Ori de câte ori este apelat un macro, codul lui este inserat în program, ceea ce va determina creşterea dimensiunilor programului. Codul unei funcţii însă apare o singură dată, indiferent de numărul de apeluri din program. Ca atare, din punct de vedere al economiei de memorie, este mai avantajos să lucrăm cu funcţii decât cu macrouri.

Pe de altă parte, folosind macrouri se câştigă timp la execuţia programului. Apelând o funcţie, trebuie să se facă o serie de pregătiri pentru a face salt la codul funcţiei şi pentru revenirea în programul apelant. Astfel, din punctul de vedere al vitezei de lucru, este mai avantajos să utilizăm macrouri în locul funcţiilor. În plus, utilizând macrouri, programul într-o primă fază poate apare mai uşor de citit, dar făcând exces de macrouri el poate deveni greu inteligibil.

În aceste condiţii, decizia când să se utilizeze un macro şi când o funcţie trebuie luată de la caz la caz şi în ultimă instanţă este o chestiune de stil de programare.

În încheiere, să mai facem o serie de observaţii privind construcţia acestor macrouri. În macrouri avem posibilitatea să utilizăm paranteze, ceea ce ne permite să evităm o serie de erori. Să presupunem că avem un program care conţine următoarele linii :

```
#define SUM(x, y) x+y;
..................
rez = 10* SUM(3, 4)
```

Ce valoare va avea variabila „rez“ ?

Conform celor spuse mai sus, preprocesorul înlocuieşte pe SUM(3, 4) cu 3+4 şi vom avea :

rez=10* 3+4

Deci, „rez" va avea valoarea 34 şi nu 70 cum pare a fi dorit programatorul.

Pentru a obţine rezultatul dorit va trebui să scriem :

#define SUM(x, y) (x+y);

şi atunci vom avea

rez=10 * (3+4)

adică rez = 70.

În general, pentru mai multă siguranţă, se recomandă să se pună paranteze atât în jurul întregului text al directivei # **define** care utilizează argumente, cât şi în jurul fiecărui argument.

19.11.3. **Directiva #include**

Directiva către preprocesor # **include** determină *includerea unui fişier sursă în alt fişier sursă*. Să presupunem că trebuie să scriem mai multe programe în care în mod repetat se calculează ariile diferitelor figuri geometrice. Formulele de calcul ale ariilor se pot plasa ca macrouri într-un fişier sursă separat, în loc să se scrie de fiecare dată macrourile utilizate în cadrul fiecărui program. Acest fişier sursă poate arăta astfel :

```
#define PI 3.14159
#define ARIA-CERC(rad) (PI*rad*rad)
#define ARIA-DREPT(lung, lat) (lung*lat)
#define ARIA-TRIUNGHI(baza, înalt) (baza*înalt/2)
#define ARIA-TRAPEZ(înalt, baza1, baza2) (înalt*(baza1+baza2)/2)
```

El se editează de exemplu cu *wordstar* şi se salvează cu extensia „.h"; poate fi numit „**areas .h**". Extensia „.**h**" (de la **header**) este standard pentru astfel de fişiere. Cînd programatorul îşi scrie programul, va plasa instrucţiunea:

#include "areas .h"

la începutul acestuia, ceea ce va determina, înainte de compilare, preprocesorul să caute fişierul „**areas .h**" în directorul curent şi să-l includă în fişierul sursă al programului. În acest fel, toate instrucţiunile de mai sus sunt adăugate în programul scris.

19.12. PROTOTIPURI PENTRU FUNCŢIILE DE BIBLIOTECĂ

Faţă de cele spuse pînă în acest moment despre funcţii, un cititor perspicace poate semnala o contradicţie în ceea ce priveşte utilizarea funcţiilor din biblioteca limbajului C în programele scrise în capitolele anterioare. Pentru funcţiile scrise de utilizator apar clar prototipurile acestor funcţii ; dar care sunt prototipurile pentru funcţiile din bibliotecă? Acestea se află grupate într-un director special — **Standard Header Directory**. De exemplu, fişierul care conţine prototipurile pentru funcţiile **printf()** şi **scanf()** se numeşte „**stdio .h**" (alături de alte prototipuri şi definiţii de funcţii şi macrouri). Pentru a include acest fişier în fişierul sursă al programului utilizatorului, trebuie să se scrie la începutul programului (înainte de **main()**):

#include <stdio .h>

În cazul utilizării funcţiei **getche()** se va pune la începutul programului înainte de **main()**:

#include <conio .h>

Semnele <> din directiva # **include** cer preprocesorului să caute fişierul header în Standard Header Directory.

Observaţie : în programele prezentate în capitolele 17—19, deşi s-au utilizat funcţii din biblioteca C, (printf(), scanf(), getche()) nu au fost trecute cu directiva #include fişierele header în care se află prototipurile şi definiţia lor. Aceste funcţii fiind foarte frecvent utilizate, unele implementări ale limbajului C determină automat preprocesorul să le caute în directorul şi în fişierele header în care se află.

Există însă şi versiuni care cer în mod obligatoriu includerea în program a fişierelor header pentru absolut toate funcţiile din bibliotecă utilizate. Fără prezenţa acestor fişiere introduse cu directiva #include, apar erori în faza de compilare a programelor generând un mesaj specific, care menţionează lipsa prototipului funcţiei din structura programului.

În consecinţă, dacă programele scrise în capitolele 17—19 dau erori la compilare de tipul celei specificate mai sus, se vor introduce în plus în programe înainte de **main()** două linii de program cu :

```
#include <stdio.h>
#include <conio.h>
```

Singura funcţie în C care nu necesită utilizarea unui prototip este funcţia **main()**. La această funcţie declaratorul ei poate fi scris complet, ca de exemplu:

```
void main(void)
```

sau cu argumente atunci cînd este cazul, aşa cum se va vedea într-unul din capitolele următoare.

19.13. FUNCŢII ŞI OPERATORI LA NIVEL DE BIT

Aceşti operatori se pot aplica numai la tipuri întregi (**char, int, short** şi **long**, cu sau fără semn). Operatorii binari sunt :

— & **(si logic la nivel de bit)**
— | **(sau logic la nivel de bit)**
— ∧ **(sau exclusiv la nivel de bit)**
— <<**(deplasare stânga)**
— >>**(deplasare dreapta)**

iar operatorul unar ~ reprezintă negare la nivel de bit (complement faţă de 1).

Trebuie distins corect între & şi &&, respectiv între | şi ||. De exemplu, 1 && 2 are valoarea 1, pe cînd 1 & 2 are valoarea 0.

Deplasarea la stînga completează dinspre dreapta cu 0, dar deplasarea la dreapta poate completa dinspre stânga fie cu 0 fie cu bitul de semn, depinzând de implementare (*shift logic* sau *shift aritmetic*). Pentru operanzi de tip *unsigned*, completarea se face totdeauna cu 0.

Se pot scrie expresii în care apare operatorul ~ aplicat unor constante, acesta fiind un bun exemplu de situaţie în care contează dimensiunea constantei. De exemplu, ~0 înseamnă întregul cu toţi biţii 1 (uzual 2 octeţi), dar ~0L înseamnă întregul lung cu toţi biţii 1 (uzual 4 octeţi).

Următoarea funcţie tipăreşte reprezentarea binară a unui întreg fără semn:

```
void bit_int(unsigned x)
{
        unsigned masca = ~ (~0U >>1);
        for(; masca !=0; masca >>=1)
                putchar ((x & masca)? '1' : '0';
```

~0U înseamnă întregul *unsigned* cu toți biții 1. Deplasarea la dreapta și tipul *unsigned* provoacă ștergerea bitului cel mai semnificativ. Negatul acestei valori va avea bitul cel mai semnificativ 1 și restul 0. Cât timp masca nu devine 0, se selectează bitul corespunzător din x (cu &) și se tipărește cifra '1' sau '0' după cum este acest bit. Apoi se deplasează „masca" la dreapta pentru a trece la următorul bit. Această procedură poate fi apelată cu orice tip de întreg (*signed* sau *unsigned*). De exemplu, pentru a vedea reprezentarea binară a lui −1 se poate scrie **bit_print(−1);**.

Dacă se schimbă tipurile lui „x" și „masca" la *unsigned long* și, totodată, se modifică constanta 0U în 0UL, se obține o procedură care tipărește reprezentarea binară a unei variabile long.

Funcția următoare realizează afișarea reprezentării binare a unui octet :

```
void bit_byte (unsigned char x)
{
    unsigned char masca = ~(~((unsigned char)'\xff')>>1);
    for (; masca; masca >>=1)
        putchar (x & masca)? '1' : '0');
}
```

Expresia (*unsigned char*) '*\xff*' înseamnă octetul cu toți biții 1, interpretat fără semn.

Iată un alt exemplu. Codul ISO pe 8 biți este similar codului ASCII (care este un cod pe 7 biți), cu diferența că bitul 7 (cel mai semnificativ) este bit de paritate (este poziționat astfel încât numărul total de biți de 1 ai caracterului să fie par). Acest cod este folosit la codificarea pe banda perforată a programelor pentru comanda numerică a mașinilor unelte. Următoarea funcție convertește un octet din cod ASCII în cod ISO:

Exemplul 19.14

```
unsigned char iso(unsigned char c)
{
    unsigned char masca='\x80';
    unsigned char nrbits=0;
    for(; masca; masca >>=1)
        if(masca & c)
            nrbits ++;
    return (nrbits & 1) ? c| '\x80':c;
}
```

Se numără efectiv numărul de biți de 1 și, dacă este impar (adică bitul 0 din „nrbits" este 1) se forțează 1 pe bitul 7 al caracterului. Funcția poate fi testată cu secvența :

```
unsigned char i;
for (i=0; i<=127; i++) {
    printf("\nASCII="); bit_byte(i);
    printf("ISO="); bit_byte(iso(i));
}
```

Următoarea funcție realizează rotația la stânga sau la dreapta a unui întreg fără semn, cu un număr dat de biți, întorcând valoarea rotită. Parametrul „sens" poate fi 'r', 'R', 'l' sau 'L', altfel x este întors nemodificat. Funcția **toupper**(c) întoarce caracterul convertit la litera mare.

Exemplul 19.15

```
unsigned rotate(unsigned x, unsigned n, char sens)
{
    unsigned masca_1, masca_2, carry;
    if((sens=toupper(sens))=='R'){
        masca_1=1;
        masca_2=~(~0U>>1);
    }
    else
        if(sens=='L') {
            masca_1=~(~0U>>1);
            masca_2=1;
        }
        else
            return x;
    while (n--) {
        carry=x & masca_1;
        (sens=='R') ? (x>>=1) : (x<<=1);
        x |= (carry) ? masca_2 : 0;
    }
    return x;
}
```

19.14. CLASE DE ALOCARE PENTRU VARIABILE

Obiectele de bază ale limbajului sunt variabilele. Variabilele se declară la începutul funcțiilor **variabile locale** sau în exteriorul tuturor funcțiilor **variabile globale**, precizându-se *clasa de alocare, tipul, numele* și o *eventuală inițializare.* Clasa de alocare determină unde se va rezerva spațiu pentru variabilă, durata ei de viață, vizibilitatea și cum se face o eventuală inițializare. Tipul determină domeniul posibil de valori și operațiile permise asupra variabilei.

Clasele de alocare principale sunt **automatic** și **static.** Variabilele din clasa automatic sunt definite în interiorul funcțiilor, fiind locale funcției în care au fost definite (nu pot fi accesate din alte funcții). *Durata lor de viață* este durata de execuție a funcției respective, aceste variabile *dispărând* când se încheie execuția funcției. De fapt, acestor variabile li se alocă spațiu în stivă, la intrarea în funcție, iar la ieșirea din funcție stiva este descărcată, deci se poate spune că variabila *„dispare"*.

Dacă nu este specificată nici o clasă de alocare, declarațiile din interiorul unei funcții creează variabile în clasa **automatic**, deci în mod implicit, toate variabilele sunt memorate în stivă. Parametrii funcțiilor sunt transmiși de asemenea prin stivă. Aceasta face ca, în C, toate funcțiile să fie în mod implicit reentrante, ceea ce permite de exemplu recursivitatea directă sau indirectă (apelul unor funcții din ele însele). Clasa **automatic** se poate specifica explicit prin cuvântul cheie **auto.**

Variabilele definite în interiorul unei funcții, cu clasa de alocare **register,** sunt asemănătoare celor din clasa **automatic,** cu excepția faptului că, dacă e posibil, ele vor fi memorate în registrele unității centrale și nu în memoria RAM. De aici rezultă unele restricții (nu există adresă de memorie asociată). Practic, numai un număr limitat de variabile pot fi ținute în registre. Dacă acest lucru nu este posibil, compilatorul alocă aceste variabile în clasa **auto,**

dar cu păstrarea restricțiilor de la clasa **register**, această comutare fiind transparentă pentru utilizator. Variabilele, parametri formali, pot fi declarate în clasa **register**, de exemplu :

```
void f(register int x);
```

aceasta neafectând modul lor de transmitere către funcție (care se face tot prin stivă), ci numai faptul că vor fi ținute în registrele mașinii pe durata execuției funcției).

Variabilele din clasele **auto** și **register** nu își păstrează valoarea de la un apel la altul al funcției în care sunt definite. Dacă sunt inițializate, inițializarea are loc la fiecare nouă intrare în funcție.

Variabilele din clasa **static** diferă de cele din clasa **auto** și **register** prin aceea că sunt memorate în locații fixe de memorie (au permanent asociată aceeași adresă). Durata lor de viață este pe parcursul execuției întregului program. O variabilă în clasa **static**, definită în interiorul unei funcții (se mai spune că este în clasa *static internal*), trebuie să aibă în față cuvântul cheie **static**, altfel ar fi definită în clasa **auto**. Pentru o variabilă în clasa **static**, inițializată, inițializarea se va produce o singură dată, la încărcarea programului (de fapt inițializarea are loc la compilare).

Variabilele definite în interiorul unei funcții (indiferent de clasă) sunt vizibile numai în funcția în care sunt definite.

Variabilele definite în exteriorul tuturor funcțiilor, fără vreun specificator de clasă, sunt în clasa **extern** și sunt totdeauna alocate la adrese fixe de memorie. Ele sunt vizibile în modulul de program în care sunt definite, de la locul de definire spre sfârșitul fișierului sursă (din acest motiv se definesc de obicei la începutul fișierului). De asemenea, sunt vizibile în toate modulele de program care compun aplicația (se spune că au „*external linkage*"). Din punctul de vedere al duratei de viață au toate caracteristicile care derivă din alocarea statică (la adrese fixe de memorie).

Dacă se dorește ca o variabilă externă să fie vizibilă numai în modulul de program în care este definită, se pune cuvântul cheie **static** (se mai spune că are clasa *static external* sau „internal linkage").

Funcțiile sunt totdeauna definite la nivelul cel mai exterior al programului și au asociate adrese fixe de memorie. Sunt implicit în clasa **external**, deci sunt vizibile în modulul în care sunt definite, de la locul de definire spre sfârșitul fișierului, ca și în toate celelalte module de program. Pentru a limita vizibilitatea numai la modulul în care sunt definite, se pune cuvântul cheie **static** (se spune că sunt în clasa **static external**). De obicei, la începutul fișierului se pune prototipul funcției.

Pentru a accesa o variabilă definită la nivel exterior în alt modul decât cel curent, se declară variabila explicit cu cuvântul cheie **extern**, care face să nu se aloce memorie pentru variabila respectivă, ci doar să se declare variabila ca fiind externă. Pentru a accesa funcții definite în alt modul (de exemplu funcții de bibliotecă) este suficientă prezența prototipului în modulul în care este folosită funcția, dar se poate declara și explicit cu **extern**. Declarația **extern** este implicită pentru variabile definite la nivel exterior și pentru funcții. Se vede că variabilele definite la nivel exterior și funcțiile sunt implicit *publice*, adică vizibile din toate modulele care compun aplicația (dacă nu au specificatorul static). Este evident că o funcție trebuie definită într-un singur modul (altfel apare eroare la *link-editare*), dar poate fi declarată în mai multe module. Similar, o variabilă **externă** trebuie inițializată doar într-un singur modul de program, altfel apare conflict la link-editare.

Pentru variabilele **auto** sau **register** inițializate, inițializarea se face la fiecare intrare în funcție (ca și cum s-ar executa o instrucțiune de atribuire). Pentru variabilele **statice** inițializate, inițializarea se face la încărcarea programului, deci o singură dată. Variabilele alocate **static** (în funcții sau la nivel exterior) neinițializate explicit, sunt inițializate implicit cu 0. Variabilele **auto** și **register** neinițializate explicit vor conține valori inițiale nedefinite.

Situațiile descrise mai sus sunt rezumate în următorul tabel :

Caracteristica → Clasa de alocare	Vizibilitate	Adresă fixă	Păstrează valoarea	Durata de viață
auto, register (variabila definită în funcție)	în interiorul funcției	Nu	Nu	Pe durata funcției
static internal (variabila definită în funcție)	în interiorul funcției	Da	Da	Pe durata aplicației
external (variabila definită la nivel exterior)	în toate modulele aplicației	Da	Da	Pe durata aplicației
static external (variabila definită la nivel exterior)	în modulul în care este definită	Da	Da	Pe durata aplicației
external (funcție la nivel exterior)	în toate modulele aplicației	Da	—	Pe durata aplicației
static external (funcție la nivel exterior)	în modulul în care este definită	Da	—	Pe durata aplicației

Iată un exemplu, cu două module compilate separat :

Exemplul 19.16

```
/*Fişier 1*/
#include <stdio.h>
int x=1;
int f(void);
void main (void)
{
      int y=2;
      static z =3;
      x=z++ +f();
      y=x++;
      printf ("main: x= %d, y din main= %d, z= %d\n", x,y,z);
      y--;
      printf ("main: x= %d, y din main= %d, z= %d\n", x,y,z);
*Fişier 2*/
#include <stdio.h>
extern int x;
```

```
int u=4;
int f(void)
{
    int w;
    static int y;
    y++;
    w=x+u++ +y;
    printf("f:x= %d, y din f= %d, u= %d, w= %d\n", x,y,u,w);
    return w;
}
```

În urma execuției, se va afișa la consolă :

```
f:x=1, y din f=1, u=5, w=6
main: x=10, y din main=9, z=4
f: x=10, y din f=2, u=6, w=17
main: x=21, y din main =8, z=5
```

Variabilele y din **main** și din f, sunt distincte. Variabila **statică** y din f, inițializată implicit cu 0, își păstrează valoarea de la un apel la altul al funcției f. Același lucru se întâmplă pentru variabilele **externe** x și u.

EXERCIȚII

1. Scrieți o funcție și un program care să ordoneze crescător trei numere întregi introduse de operator de la tastatură.

2. Scrieți un program care să permită introducerea succesivă de la tastatură a n numere reale și să afișeze la sfârșit care a fost cel mai mic și cel mai mare număr introdus. Pentru calculul minimului și maximului celor n numere se vor folosi două funcții.

3. Să se scrie un program pentru rezolvarea unui sistem de ecuații algebrice liniare de ordinul 2. Se vor analiza toate cazurile. Pentru calculul determinantului se va folosi un **macro**.

4. Să se scrie un program care să calculeze numărul e cu poziția **ε** (introdusă de operator de la consolă). Se va aplica formula :

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \ldots + \frac{1}{n!}$$

Pentru calculul factorialelor și al diferitelor aproximații ale numărului e se va folosi o funcție. Considerăm că numărul este calculat cu precizia dorită dacă $|e_{n+1} - e_n| \leq \varepsilon$ unde e_{n+1} și e_n sunt două aproximații succesive ale numărului e.

CAPITOLUL 20
TABLOURI ŞI POINTERI

CAPITOLUL 20

TABLOURI ŞI POINTERI

Pointerii sunt *variabile care conţin adresele unor alte variabile* sau, mai general, *ale unor obiecte din memorie*. În limbajul C există o strânsă legătură între tablouri şi pointeri, de aceea aceste două tipuri de date vor fi prezentate împreună. Tablourile şi pointerii sunt primele două exemple de **tipuri derivate,** deci de tipuri obţinute pe baza tipurilor simple de date, prin construcţii specifice limbajului.

20.1. TABLOURI CU O DIMENSIUNE

Un tablou cu o dimensiune este, prin definiţie, o succesiune de variabile de acelaşi tip (numit **tipul de bază** al tabloului) care ocupă o zonă continuă de memorie, deci elementele tabloului se află în memorie la adrese succesive. Un tablou are asociată o **dimensiune,** care este numărul de elemente al tabloului şi un **nume,** care identifică global tabloul. De asemenea, ca orice obiect în limbajul C, tabloul are asociată şi o **clasă de alocare,** după locul şi modul în care este definit.

Declaraţia unui tablou pune în evidenţă toate aceste caracteristici şi are forma generală:

clasa T nume[N];

unde **clasa** este clasa de alocare (care poate lipsi, caz în care se presupune o clasă implicită, exact ca la variabile simple), **T** este tipul, **nume** este numele tabloului, iar **N** este o constantă întreagă specificând dimensiunea.

Exemple de definiţii de tablouri sunt :

```
int a[20];
char sir[80];
float x[5];
```

Declaraţiile de mai sus spun că **a** este un tablou de **20** de variabile de tip **int, sir** este un tablou de **80 de caractere,** iar **x** este un tablou de **5** variabile de tip **float.**

Elementele tabloului pot fi accesate prin numele tabloului urmat de o expresie întreagă cuprinsă între paranteze drepte. Expresia trebuie să ia valori între **0** şi **dimensiunea tabloului minus 1.** De cele mai multe ori, aceste

expresii sânt constante sau variabile simple şi se numesc **indici.** De exemplu, referiri corecte la elementele tabloului **a** de mai sus sînt **a[0]**, **a[1]**, ..., **a[19]**, iar elementele tabloului **x** sunt **x[0]**, **x[1]**, ..., **x[4]**.

Tablourile pot fi declarate în interiorul funcţiilor, caz în care sunt implicit în clasa de alocare **auto**, sau în exteriorul funcţiilor, caz în care sunt implicit în clasa **extern** (sunt alocate la adrese fixe şi vizibile în tot cuprinsul modulului curent, ca şi în toate modulele care constituie aplicaţia). Un tablou declarat în interiorul unei funcţii, cu specificatorul de clasă **static**, va fi alocat la adrese fixe de memorie, dar va fi vizibil (accesibil) doar în funcţia respectivă. Un tablou declarat în exteriorul funcţiilor, cu specificatorul de clasă **static** va fi alocat la adrese fixe, dar va fi vizibil numai în modulul curent de program, nu şi în alte eventuale module. Tablourile nu pot fi în clasa **register.**

Prelucrările de tablouri se implementează de obicei prin cicluri **for**, deoarece se cunoaşte aprioric numărul de iteraţii. Iată o secvenţă de program care afişează la consolă un tablou de **n** întregi, într-o formă „frumos aliniată":

```
#define N 100
int a[N];
int i;
for (i=0; i < N; i++)
        printf ("%6d%c", a[i], ((i%10)==9 || i==N-1) ?'\n':' ');
```

Expresia care corespunde specificatorului de format %*c* este o expresie condiţională care se evaluează fie la **'\n'** fie la ' ', după cum s-a tipărit tot al 10-lea caracter sau ultimul.

În ceea ce priveşte iniţializarea tablourilor, aceasta respectă regulile care derivă din clasa de alocare : un tablou în clasa **auto** (iniţializat explicit) va fi iniţializat la fiecare intrare în funcţia respectivă, iar un tablou în clasa **static** sau **external** (iniţializat explicit) se iniţializează o singură dată, la încărcarea programului în memorie. Tablourile în clasa **static** sau **extern**, neiniţializate explicit, sunt iniţializate implicit cu 0, iar cele în clasa **auto** nu sunt iniţializate, deci vor conţine valori nedefinite (oarecare). Tablourile în clasa **auto** se pot iniţializa numai cu constante.

Iniţializarea se face prin semnul = după declaraţia tabloului urmat de o listă de valori iniţiale, separate prin virgule, care se include între acolade. De **exemplu:**

```
int a[4]={1, 2, 3, 4};
char s[4]={'a', 'b', 'c', '\0'};
```

Dacă lista de valori iniţiale cuprinde mai puţine date decât dimensiunea declarată a tabloului, celelalte valori se iniţializează cu 0, indiferent de clasa de alocare. De exemplu, în urma declaraţiei:

```
int a[10]={5, 7, -1};
```

elementele a[3], a[4], ..., a[9] vor conţine valoarea 0.

În cazul unui tablou iniţializat, se poate omite dimensiunea, caz în care ea este calculată implicit de către compilator, după numărul de elemente din lista de iniţializare. De exemplu, declaraţia :

```
float x[]={1.0, 2.0, 3.0};
```

spune că **x** este un tablou cu 3 elemente, acestea având valorile inițiale date de lista respectivă.

Un caz special îl reprezintă *tablourile de caractere*, care pot fi inițializate și cu șiruri constante de caractere (incluse între ghilimele). Declarația.

```
char s[]="abcdef";
```

este perfect echivalentă cu:

```
char s[]={'a', 'b', 'c', 'd', 'e', 'f', '\0'};
```

observîndu-se prezența caracterului special **'\0'** octetul nul, care acționează ca terminator de șir).

Totuși, dacă este specificată explicit dimensiunea tabloului de caractere și aceasta coincide cu numărul de caractere din șirul constant, terminatorul **'\0'** nu se mai include. Declarația:

```
char s[6]="abcdef";
```

este echivalentă cu :

```
char s[6]={'a', 'b', 'c', 'd', 'e', 'f'};
```

De obicei, tablourile de caractere se inițializează astfel încât ultimul caracter să fie **'\0'**, acest fapt înlesnind funcțiile de prelucrare

Transmiterea tablourilor cu o dimensiune la funcții se face declarând parametrul formal al funcției ca fiind tablou, lucru care se realizează prin prezența parantezelor drepte. Nu este necesară prezența explicită a dimensiunii tabloului între paranteze, deoarece ceea ce se transmite efectiv la funcție este adresa de început a tabloului. Dacă este necesar, dimensiunea tabloului se precizează într-un parametru formal separat.

Iată un **exemplu** de *funcție care calculează și întoarce* **produsul scalar al doi vectori** *de dimensiune* **n**, *reprezentați prin două tablouri de numere reale (float)*:

```
floatprod_scad (float x[], float y[], int n)
{
        float prod=0.0;
        int i;
        for (i=0; i<n; i++)
                prod += x[i] * y[i];
        return prod;
}
```

Dacă sunt declarate două asemenea tablouri, de dimensiune 3:

```
float a[3]={-1.0, 2.0, 2.5};
float b[3]={0.0, 4.0, -6.0};
float ab;
```

atunci produsul scalar al vectorilor **a** și **b** se poate obține prin apelul :

```
ab=prod_scal(a, b, 3);
```

Se vede deci că, în apel, se precizează numai numele tablourilor respective.

Un alt **exemplu** îl constituie **calculul produsului vectorial** *al doi vectori daţi, de dimensiune 3 (vectori euclidieni)*. După cum se ştie, produsul vectorial este tot un vector şi nu poate fi întors direct de funcţie, deoarece funcţiile pot întoarce numai tipuri simple de date. Putem însă declara tabloul rezultat (produsul vectorial) tot ca parametru al funcţiei. Regula de calcul a produsului vectorial al vectorilor x şi y este dată de dezvoltarea formală după prima linie a determinantului:

$$\begin{vmatrix} i & j & k \\ x[0] & x[1] & x[2] \\ y[0] & y[1] & y[2] \end{vmatrix}$$

în care **x[0]**, **x[1]**, **x[2]** sînt coordonatele vectorului **x**, şi similar pentru **y**, iar **i**, **j** şi **k** sunt versorii axelor de coordonate.

```
void prod_vec (float x[], float y[], float pr_vec[])
{
      pr_vec[0]=x[1]*y[2]—x[2]*y[1];
      pr_vec[1]=—x[0]*y[2]+y[0]*x[2];
      pr_vec[2]=x[0]*y[1]—x[1]*y[0];
}
```

Funcţia se poate folosi declarând convenabil trei tablouri:

```
float a[3]={—1.0, 2.0, 2.5};
float b[3]={0.0, 4.0, —6.0};
float a_X_b[3];
prod_vec(a, b, a_X_b);
```

Tablourile de caractere se prelucrează într-un mod uşor diferit, deoarece ele nu au de obicei specificată dimensiunea, ci aceasta se deduce implicit din prezenţa terminatorului **'\0'**, care este ultimul caracter din tabloul (şir). O funcţie care calculează lungimea (numărul de caractere) dintr-un asemenea tablou este:

```
int lng_sir (char s[])
{
      int i;
      for (i=0); s[i] !='\0'; i++)
            ;
      return i;
}
```

Se observă că terminatorul **'\0'** nu se numără (nu se consideră caracter util). Această funcţie este analogul funcţiei de bibliotecă **strlen**. Un exemplu de apel este;

```
char s[]="1234567890";
int n;
n=lng_sir(s);
```

O altă situaţie tipică de prelucrare a unui tablou este **căutarea** *unui element dat într-un tablou dat*. Funcţia de căutare întoarce de obicei indicele primei apariţii a obiectului, sau —1 dacă obiectul nu apare în tablou. Căutarea se numeşte în acest caz *liniară* şi se implementează în felul următor (presupunem un tablou de întregi):

```
int cautä_lin(int tab[], int obiect, int n)
{
    int i;
    for (i=0; i<n; i++)
        if(obiect ==tab[i])
            return i;
    return -1;
}
```

Se parcurg elementele tabloului și, dacă se găsește un element egal cu obiectul căutat, se întoarce programului apelant indicele său. Dacă s-a ajuns la sfârșitul buclei **for**, înseamnă că obiectul nu este în tablou și atunci se întoarce −1.

Este evident că, în medie, acest algoritm este de ordin **n**, adică se efectuează un număr de comparații proporțional cu **n**.

O îmbunătățire serioasă a algoritmului se poate face în cazul **căutării** *unui obiect într-un tablou cu elemente distincte, ordonate crescător*. În acest caz algoritmul se numește de **căutare logaritmică** (sau **binară**) și este asemănător cu metoda înjumătățirii intervalului la calculul aproximativ al rădăcinii unui polinom. Se compară obiectul căutat cu elementul de la mijlocul tabloului și, funcție de rezultatul comparației, se localizează obiectul în una din cele două jumătăți. Procedeul continuă până când se găsește obiectul sau până când, prin înjumătățiri succesive, se epuizează elementele tabloului. Implementarea este următoarea:

```
int cautä_bin(int tab[], int obiect, int n)
{
    int comp, stinga=0;, mijloc, dreapta=n—1;
    while (stinga<=dreapta) {
        mijloc = (stinga + dreapta)/2;
        if ((comp=obiect—tab[mijloc])<0)
                dreapta=mijloc—1;
        else
                if (com >0)
                        stinga=mijloc+1;
        else
                        return mijloc;
    }
    return —1;
}
```

Ideea este că dacă **obiect<tab[mijloc]**, atunci obiect se poate găsi numai în intervalul de indici **[stînga, mijloc-1]**, iar dacă **obiect>tab[mijloc]**, se va găsi eventual în interiorul **[mijloc+1, dreapta]**. Numărul de iterații nu poate depăși $[\log_2(n)]+1$, deoarece la fiecare iterație, dimensiunea subtabloului în care se caută se reduce la jumătate.

EXERCIȚII

1. Scrieți o funcție de tipărire a unui tablou de variabile **float**, având ca parametri formali numele tabloului și dimensiunea sa. Tipărirea trebuie să se realizeze pe câmpuri de 11 poziții, cu 3 cifre după punctul zecimal, câte 6 elemente pe un rând.

2. Scrieți o funcție care să realizeze citirea de la consolă a unui tablou de întregi. Funcția are ca parametri numele tabloului și dimensiunea sa și trebuie să afișeze pe câte o linie indicele elementului care urmează a fi citit.

3. Scrieţi o funcţie care să afişeze la consolă un şir (tablou) de caractere, considerat terminat cu '\0'. Afişarea unui caracter se poate face cu funcţia de bibliotecă **putchar**. Un apel de forma **putchar(c)** va afişa caracterul dat de variabila c. Terminatorul '\0' nu trebuie afişat, iar după ultimul caracter afişat trebuie să se afişeze un '\n'. Această funcţie este analogul funcţiei de bibliotecă **puts**.

4. Scrieţi o funcţie care să iniţializeze un tablou de întregi cu valori aleatoare. Pentru generarea de numere aleatoare întregi se pot folosi funcţia de bibliotecă **randomize** (fără parametri) care se apelează o singură dată şi iniţializează generatorul şi funcţia **random** (cu parametru **n** şi tip (întreg), care va genera aleator o valoare între **0** şi **n—1** (o instrucţiune de forma **a[i]=random(n)** va atribui lui **a[i]** o valoare aleatoare între **0** şi **n—1**).

*5. Scrieţi o funcţie care să realizeze sortarea în ordine crescătoare a elementelor unui tablou de numere întregi. Funcţia va avea ca parametri numele tabloului şi dimensiunea sa. Algoritmul de sortare (unul dintre cele mai simple, dar şi dintre cele mai ineficiente) este următorul (se presupune numele tabloului **a**):

```
sortat =0;
cât timp sortat ==0 {
    sortat =1;
    pentru i=0 până la n-2 cu pasul 1 {
        dacă a[i]>a[i+1]{
            interschimbă a[i] cu a[i+1]
            sortat =0;
        }
    }
}
```

Scrieţi un program de test care să afişeze un tablou, apoi să-l sorteze şi să-l afişeze din nou. Scrieţi două variante de program de test în care dimensiunile tabloului să fie 50 şi 500 de întregi. Pentru iniţializarea tabloului se poate folosi funcţia din exerciţiul 20.4, iar pentru afişare o funcţie similară cu cea din exerciţiul 20.1.

Ce puteţi spune despre timpul de execuţie al funcţiei de sortare în cele două cazuri?

Încercaţi să îmbunătăţiţi algoritmul de sortare, eliminând comparaţiile redundante şi observaţi efectul asupra timpului de execuţie.

20.2. POINTERI. DECLARAREA POINTERILOR

Pointerii sunt *variabile care conţin* (sunt capabile să conţină) *adresele unor alte variabile sau obiecte*, deci practic *adrese de memorie*. Dimensiunile pointerilor (câţi octeţi ocupă) depind de maşină, de implementarea limbajului, etc. În general, acest aspect trebuie luat în consideraţie doar din punct de vedere al portabilităţii programelor; un program bun nu trebuie să depindă de dimensiunile concrete ale pointerilor şi, în general, să funcţioneze corect pe orice implementare a limbajului care respectă standardul ANSI.

Un pointer este asociat unui tip de variabile : vom avea pointeri către **char**, **int**, **float** etc. În general, dacă **T** este un tip de date (standard sau definit de utilizator), un pointer către tipul **T** se declară prin:

```
T *p;
```

Iată exemple de declarații de pointeri:

```
char    *pc;
int     *pi;
float   *pf;
```

care spun că **pc** este pointer către **char**, **pi** este pointer către **int** etc. Tipul de bază al declarației este **T**, deci, pentru a declara mai mulți pointeri în aceeași linie, se scrie * la fiecare:

```
int *p, *q, *r, i, j;
```

Aici **p**, **q**, **r** sunt pointeri la **int**, iar **i** și **j** sunt variabile de tip **int**.

Se introduc doi operatori noi, anume **operatorul de referențiere (adresare)** & și **operatorul de dereferențiere (indirectare)** *.

Operatorul de *adresare* & se aplică unei variabile, producând (furnizând) adresa variabilei. Dacă variabila este de tip **T**, atunci operatorul & întoarce tipul de date **T** *, adică **pointer** către **T**. Astfel, operatorul & este un prim mod (și cel mai important) de a obține adrese de variabile (obiecte).

Este deci corectă operația de atribuire de mai jos:

```
char c, *pc;
pc=&c;
```

Se declară variabila **c** de tip **char** și variabila **pc** de tip **pointer** către **char**; în urma atribuirii, **pc** va conține adresa variabilei **c**.

Evident că operatorul & nu se poate aplica unei variabile în clasa **register** (registrele procesorului nu au asociate adrese de memorie) și nici unei constante.

Accesul la o variabilă (obiect) prin intermediul pointerilor se face cu operatorul de **indirectare** *. Dacă **p** este un **pointer de tip T** *, atunci * **p** este prin definiție **obiectul de tip T, aflat la adresa p.**

În exemplul de mai sus, * **pc** este variabilă de tip **char**, aflată la adresa **pc**, adică tocmai variabila **c**. Avem astfel două moduri de acces la variabila **c**: prin specificarea numelui ei și prin intermediul pointerului **pc**, care a fost „poziționat" pe **c**.

Este evident că cei doi operatori, & și * sunt operatori inverși: dacă **x** este de tip **T1** și **px** este de tip **T2** *, atunci * (&**x**)este identic cu **x** și &(* **px**) este identic cu **px**.

● **În concluzie:**

— *pointerii reprezintă adrese ale unor zone de memorie;*

— *putem avea acces la acele zone, prin operatorul* *;

— *dimensiunea și semnificația unei zone indicate de un pointer depinde de tipul pointerului.*

● Ca **regulă generală** despre **ce este permis și ce nu cu un pointer** să reținem că:

dacă pointerul p indică o variabilă x, atunci expresia * p poate apare în orice context în care este permisă apariția lui x.

Iată un **exemplu** în care un *pointer indică obiecte diferite pe parcursul execuției programului:*

```
void main (void)
{
    int i=1, j=5, *p=&i;
    *p=2;
    (*(p=&j))++;
    printf ("%d %d\n", i, j);
}
```

Se declară variabile întregi **i** și **j**, inițializate cu 1 și, respectiv cu 5, și o variabilă **p** tip pointer către **int**, inițializată cu adresa lui **i**. Prin intermediul lui **p**, se atribuie practic lui **i** valoarea 2 (***p=2;**) și apoi se atribuie lui **p** adresa lui **j**, după care se incrementează conținutul lui **p**, adică variabila **j**. Astfel, programul va tipări valorile 2 și 6.

O folosire importantă a pointerilor este **transferul de referință al parametrilor unei funcții.** În C, transferul implicit este prin valoare, așa cum s-a precizat la funcții. Dacă vrem ca o funcție să modifice o variabilă parametru formal, trebuie să transmitem funcției adresa variabilei, iar în interiorul funcției să folosim operatorul de indirectare.

Exemplul clasic este o *funcție de interschimbare a două variabile*. O primă încercare ar fi:

```
void schimbă_1 (int a, int b)
{
    int temp;
    temp=a; a=b; b=temp;
}
```

Dacă se apelează această funcție cu:

```
int x=1, y=2;
schimbă_1(x, y);
```

se constată că variabilele **x** și **y** rămân nemodificate. Explicația este că funcția lucrează cu variabilele ei locale **a** și **b**, pe care le interschimbă, acest lucru nereflectându-se asupra parametrilor actuali **x** și **y**. Soluția corectă este:

```
void schimbă_2(int *p, int *q)
{
    int temp;
    temp=*p; *p=*q; *q=temp;
}
```

iar apelul corespunzător va fi:

```
int x=1, y=2;
schimbă_2(&x, &y);
```

Variabilele locale **p** și **q** primesc acum adresele lui **x** și **y**, iar prin ***p** și ***q** vom accesa chiar pe **x** și **y**. Cele două situații sunt descrise în figura 20.1.

EXERCIȚII

6. Scrieți un program principal în care declarați câteva variabile întregi și un pointer către **int**, atribuind succesiv pointerului adresele variabilelor întregi și afișând pointerul și întregul indicat de pointer, după fiecare atribuire. Afișarea unui pointer se face cu funcția **printf**, cu specificatorul de format %!.

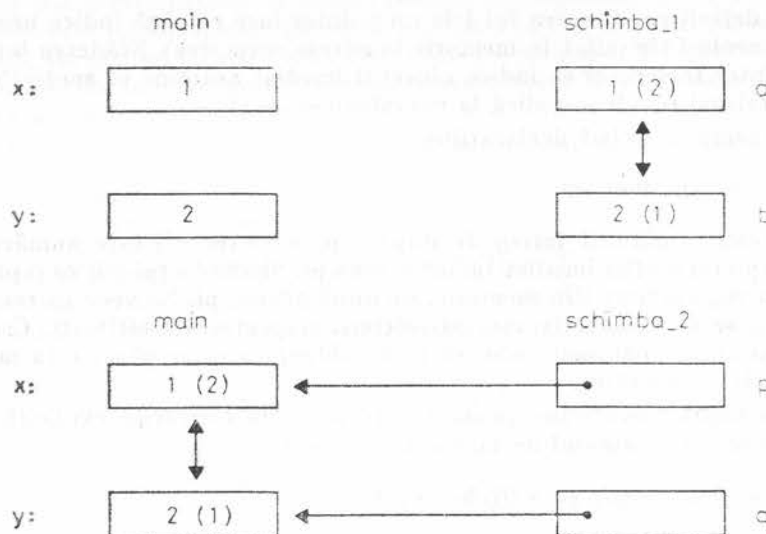


Fig. 20.1.

7. Precizaţi ce tipăreşte următorul program (fără a-l rula). Verificaţi apoi răspunsul prin execuţia propiu-zisă a programului.

```
#include <stdio.h >
int f(int *p)
{
    return (*p)++;
}
void main(void)
{
    int x=1, y=2;
    y=f( &x); printf("x=%d y=%d\n", x, y);
    y=f(&x); printf ("x=%d y=%d\n", x, y);
}
```

8. Acelaşi lucru pentru programul:

```
#include <stdio.h>
int f(int a, int*p)
{
    a=— —(*p);
    return a;
}
void main (void)
{
    int x=1, y=2;
    y=f(y, &x); printf ("x=%d y=%d\n", x, y);
    y=f(y, &x); printf("x=%d y=%d\n", x, y);
```

20.3. ARITMETICA POINTERILOR

Operaţiile aritmetice permise asupra pointerilor sunt: adunarea/scăderea unei constante, incrementarea/decrementarea şi scăderea a doi pointeri de acelaşi tip.

Prin definiție, adunarea lui 1 la un pointer face ca el să indice **următorul obiect de același tip** (aflat în memorie la adrese succesive). Scăderea lui 1 dintr-un pointer face ca el să indice obiectul imediat anterior în spațiul de memorie. Aceleași reguli se aplică la operatorii ++ și −−.

De exemplu, având declarațiile:

```
int *pi; float *pf;
```

***(pi+1)** este următorul **întreg** de după ***p**, iar ***(pf−1)** este **numărul real în simplă precizie** aflat imediat înaintea lui ***pf**. Similar ***(pi+i)** va reprezenta al **i**-lea întreg succesiv din memorie, de după adresa **pi**. Se vede că ceea ce se adună sau se scade efectiv este **sizeof(int)**, respectiv **sizeof(float)**. Cade evident în sarcina programatorului să țină evidența a ce se găsește în memorie la adresele respective.

Semnificația pointerilor poate fi alterată prin conversie explicită de tip (**cast**). Astfel, în exemplul de mai sus, expresiile:

*((char *)pf) și *(((char *)pf)+1)

vor furniza primul și, respectiv, al doilea octet din reprezentarea unei variabile de tipul **float**. Tipul expresiei **(char *) pf** este **char *** și adunarea lui 1 la această expresie va avansa pe **pf** cu 1 octet.

Atribuirile de pointeri trebuie să se facă între pointeri (sau expresii) de același tip. Se poate folosi conversia explicită de tip, în cazul unor pointeri de tip diferit, dar acest lucru va produce, în general, programe depinzând de implementare, ceea ce nu se recomandă. De exemplu, cu variabilele declarate mai sus, putem atribui:

```
pf=(float*)pi;
```

dar această atribuire ar fi defectuoasă pe mașinile la care există *restricții de aliniere a variabilelor în memorie* (de exemplu, variabilele **float** să se afle în memorie la adrese multiplu de 4). În general, asemenea atribuiri trebuie folosite cu grijă. Limbajul permite folosirea pointerilor de tip **void ***, pe post de pointeri universali (către „*orice*"). Se garantează ca un pointer de acest tip poate fi atribuit unui alt pointer și reciproc fără probleme. Același lucru este valabil pentru pointerul **0**, care poate fi atribuit oricărui tip de pointer. Pointerul **0** este predefinit ca **NULL** și se consideră că „*nu indică nimic*", fiind folosit ca terminator în diverse structuri de date.

Deoarece **void *** înseamnă de fapt pointer de tip neprecizat, nu putem face operații aritmetice direct asupra acestor pointeri, deoarece nu se cunoaște dimensiunea și domeniul lor. De asemenea, nu putem dereferenția un pointer de tip **void ***, din același motiv. În aceste cazuri, pointerul de tip **void *** trebuie convertit în prealabil la un pointer „concret". Folosirea pointerilor **void *** permite scrierea unor funcții cu grad maxim de generalitate. Un **exemplu clasic** este o **funcție de copiere** *a unei zone de memorie în altă zonă, care să funcționeze corect în orice situație.* Funcția primește adresele zonelor sursă și destinație și dimensiunea în octeți a zonei.

```
void moveb (void *dest, void *sursa, size_t dimb) {
    while(dimb−−)
        *((char *) dest)++=*((char*) sursa)++;
}
```

Tipul **size_t** este definit în limbaj pentru a compatibiliza diverse implementări. El este un tip întreg, garantat a fi întors de operatorul **sizeof**. Pointerii **dest** și **sursa** sunt de tip **void ***, deci funcția va putea fi folosită cu orice tip de adrese. În funcție, pointerii sunt convertiți explicit la **char ***, ceea ce înseamnă acces la nivel de octet.

Dacă T este un tip de date (standard sau definit de utilizator) apelul:

```
T a, b;
moveb (&a, &b, sizeof(T));
```

va copia obiectul **b** în obiectul **a**, indiferent de tipul **T**.

Am putea gândi acum o funcție la care transferul intern să aibă loc la nivel de întreg (totdeauna întregii sunt pe un număr par de octeți, de obicei 2 sau 4):

```
void movew(void *dest, void *sursa, size_t dimw)
{
    while (dimw--)
        *((int *) dest)++=*((int *)) sursa++;
}
```

La mașinile cu magistrală de 16 biți și cu set de instrucțiuni adecvat, **movew()** va face un număr de accese la memorie de 2 ori mai mic decât **moveb()**.

Putem acum scrie o funcție generală **move()**, care să țină seama de aceste lucruri:

```
void move (void *dest, void *sursa, size_t dim)
{
    size_t dimw=dim/sizeof(int);
    size_t dimb=dim % sizeof(int);
    if(dimw) {
        movew(dest, sursa, dimw);
        (int *) dest+=dimw;
        (int *) sursa+=dimw;
    }
    if (dimb)
        moveb(dest, sursa, dimb);
}
```

Aritmetica unui pointer (adunări/scăderi de constante) este garantată (în standardul ANSI C) a funcționa corect numai dacă pointerul se menține între limitele unui tablou declarat de variabile. Este permisă și poziționarea pointerului pe elementul imediat următor ultimului element al tabloului. Această restricție este importantă și provine de la mecanismele de formare a adresei fizice de memorie (segmentare).

În ceea ce privește scăderea a doi pointeri, standardul ANSI C permite această operație numai între pointeri de același tip, care să indice elemente ale unui același tablou. Dacă **p** și **q** sunt doi pointeri de tip **T ***, respectând aceste condiții și **p** indică un element al tabloului aflat în memorie „după" **q** (adică la o adresă mai mare), atunci diferența **p**−**q** reprezintă **numărul de obiecte de tip T** aflate între **p** și **q**. Există tipul întreg predefinit **ptrdiff-t** pentru asemenea diferențe. Se vede deci că întreaga aritmetică a pointerilor de tip **T *** se face considerind unitatea ca fiind **sizeof(T)**.

În aceleași condiții ca la scădere, se pot face comparații cu operatorii ==, !=, <, >, <= sau >=. Prin definiție, **p>q** este 1, dacă se poate face diferența **p — q** (conform restricțiilor de mai sus) și această diferență este pozitivă. Este permisă comparația oricărui tip de pointer cu pointerul **NULL**.

Doi pointeri nu se pot niciodată aduna.

EXERCIȚII

9. Scrieți un program principal care să ilustreze aritmetica pointerilor de tip **float**, definind un tablou de variabile de tip **float** și un pointer către **float**, inițializat cu adresa primului element al tabloului. Incrementați de câteva ori acest pointer (fără a depăși dimensiunea tabloului) și tipăriți-l cu **printf** (cu descriptorul de format %p). Cum variază adresele prin incrementarea pointerului ?

Modificați programul schimbând numai tipul tabloului și al pointerului (**int** în loc de **float** și apoi **char** în loc de **int**). Ce se schimbă în modul de variație al adreselor ?

Deduceți din acest exercițiu dimensiunea în octeți a tipurilor **float, int** și **char**, în implementarea limbajului C cu care lucrați.

10. Scrieți o funcție care să schimbe două obiecte de dimensiuni oarecare din memorie. Funcția primește adresele (de tip **void** *) ale celor două obiecte și dimensiunea lor comună (număr de octeți). Scrieți un program principal care să testeze această funcție în diverse cazuri (schimbări de întregi, de variabile reale etc.).

11. Studiați ce tipărește programul următor :

```
#include <stdio.h>
long a[10]={10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
void main (void)
{
      long *pi;
      for (pi= &a[0]; pi < &a[10; pi++)
            printf("Adresa:%p Elementul: %ld", pi,*pi);
}
```

Comparați bucla **for** din acest exercițiu cu bucla **for** obișnuită de tipărire a elementelor tabloului **a**.

20.4. POINTERI ȘI TABLOURI CU O DIMENSIUNE

Din aritmetica pointerilor rezultă multe asemănări cu tablourile cu o dimensiune. Orice operație care se poate face cu variabile indexate (cu indici) se poate face și cu pointeri și reciproc. Să considerăm declarația:

```
int a[10], *pa;
```

care definește un tablou de 10 întregi, aflați la adrese succesive de memorie și un pointer către **int**. Referirile la elementele tabloului se fac prin:

```
a[0] a[i]...a[9]
```

a[i] fiind al i-lea element al tabloului **a**.

Cum **a[i]** este o variabilă de tip int, are sens atribuirea:

```
pa= &a[0];
```

în urma căreia **pa** va conține adresa lui **a[0]**, deci adresa de început a tabloului (fig. 5.2). Expresia * **pa** va fi acum o referire corectă la **a[0]**, de exemplu:

x = * pa ;

va copia in x pe a[0]. În mod similar, *(pa+1), *(pa+2) vor fi, conform cu aritmetica pointerilor la int, referiri corecte la a[1], a[2], etc. Aceste construcții nu depind de tipurile tabloului și al pointerului, cu condiția ca aceștia să fie de același tip.

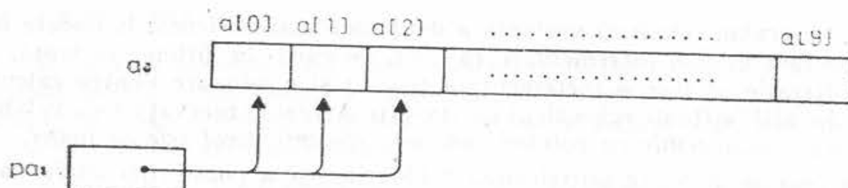


Fig. 20.2.

Acum intervine o convenție foarte importantă în C, și anume că, prin definiție, valoarea unei variabile sau expresii de tip tablou este adresa primului element al tabloului. Cu alte cuvinte, **numele tabloului este sinonim cu adresa elementului 0 al acestuia**, adică a și &a[0] reprezintă același lucru.

Atunci, atribuirea pa=&a[0] se mai poate scrie pa=a; iar dacă **a** este adresa lui **a[0]**, atunci **a+i** va fi adresa lui **a[i]** și aceasta indiferent de tipul tabloului.

Similar, conținutul de la adresa **a+i**, adică ***(a+i)**, este chiar a[i]. Se vede deci că expresiile **a[i]** și ***(a+i)** sunt perfect similare.

Dacă în identitatea **a[i]==*(a+i)** aplicăm operatorul &, obținem &a[i]==a+i
iar pentru i=0, se regăsește &a[0]==a.

Echivalențele pot continua: dacă **pa** este un pointer, atunci **pa[i]** este sinonim cu ***(ap+i)** și așa mai departe.

De fapt, compilatorul convertește orice expresie cu indici într-o expresie similară cu pointeri, fiind garantată echivalența sintactică dintre expresiile **e1[e2]** și ***((e1)+(e2))**, unde **e1** și **e2** sunt expresii valide. De aici decurge **observația interesantă** că *operația de indexare este comutativă*: **e1[e2]** este același lucru cu **e2[e1]**, fapt cu totul neobișnuit în alte limbaje.

Există totuși o distincție între pointeri și tablouri. *Pointerii sunt variabile*, deci putem să-i modificăm (**pa=a;** și **pa++;** sunt instrucțiuni corecte). *Un nume de tablou* **nu** *este o variabilă*, deci instrucțiuni ca **a=pa;** și **a++;** sunt **incorecte**.

Când un tablou este transmis ca parametru la funcție, ceea ce se transmite este adresa de început a tabloului. Funcția poate fi declarată ca având parametru formal de tip pointer. Tot aici se vede de ce nu este necesară specificarea dimensiunii tabloului, atât în funcție, cît și în apel.

Iată două variante ale funcției **strlen()**, care calculează *lungimea unui șir de caractere*:

```
int strlen(char s[])
{
    int i;
    for(i=0; s[i]!='\0'; i++)
        ;
    return i;
}
```

```
int strlen (char *s)
{
    char *p;
    for(p=s; *p!='\0'; p++)
        ;
    return p-s;
}
```

Din punctul de vedere al programului care apelează funcția, este irelevant dacă ea este scrisă cu pointeri sau cu tablouri: apelul este același:

```
char a[80];
n=strlen(a);
```

Să observăm totuși că varianta a doua este mai eficientă: la fiecare iterație, se face doar o incrementare (**p++**), pe când în prima variantă, la fiecare iterație se face o incrementare (**i++**) și o adunare pentru calculul adresei lui **a[i]** (**a[i]** este echivalent cu ***(a+i)**). Această observație este valabilă în general: *construcțiile cu pointeri sunt mai eficiente decât cele cu indici.*

În ceea ce privește inițializarea tablourilor și a pointerilor către **char**, există unele particularități. Când se inițializează un tablou de caractere, se poate specifica un șir, inclus între ghilimele, de exemplu:

```
char s[]="abcd";
```

Această declarație este absolut similară cu:

```
char s[]={'a', 'b', 'c', 'd', '\0'};
```

rezervându-se spațiu adecvat (în cazul de față 5 octeți).

Pointerii către **char** se pot inițializa într-o formă similară, dar cu semnificație diferită:

```
char *p="abcd";
```

În acest caz, compilatorul rezervă spațiu pentru șirul „**abcd**" „undeva" la o adresă fixă de memorie, inițializează acest spațiu cu caracterele respective (inclusiv terminatorul '\0') și inițializează pointerul **p** cu adresa acestui spațiu de memorie. Nu este în nici un caz vorba despre o *atribuire de șiruri*, operație care *nu există în C*, ci despre o *atribuire de pointeri*. Clasa variabilelor pointer acționează asupra inițializării în mod obișnuit: în clasa **static** inițializarea se face o singură dată, la încărcarea programului, iar în clasele **auto** și **register** la fiecare intrare în funcția respectivă.

Iată alte două **exemple de funcții clasice** pentru *operații cu șiruri de caractere*: copiere de șir în alt șir și comparație lexicografică, în variante cu tablouri și cu pointeri:

```
void strcpy (char p[], char q[])        void strcpy (char *p, char *q)
{                                       {
    int i;                                  while (*p++=*q++)
    for (i=0; p[i]=q[i]; i++)                   ;
        ;
}                                       }
int strcmp(char s[], char t[])          int strcmp (char *s, char *t)
{                                       {
    int i;
    for (i=0; s[i]==t[i]; i++)              for (; *s==*t; s++, t++)
        if (s[i]=='\0')                         if (*s=='\0')
            return 0;                               return 0;
    return s[i]-t[i];                       return *s-*t;
                                        }
```

În ambele exemple, nu s-a mai scris explicit comparaţia cu '\0', deoarece este redundantă (în C, orice valoare diferită de 0 este considerată ca valoare logică 1).

Funcţia **strcmp()** întoarce 0 dacă şirurile coincid sau diferenţa dintre primele caractere care nu coincid.

EXERCIŢII

12. Scrieţi o funcţie **strcat** care primeşte doi pointeri la **char** (adrese de şiruri de caractere terminate cu '\0') şi care realizează concatenarea (alipirea) celui de-al doilea şir la primul şir. Verificaţi că funcţia se execută corect şi în cazul în care cel puţin unul din şiruri este vid (deci primul caracter este chiar terminatorul '\0').

13. Rescrieţi funcţia de căutare logaritmică din subcapitolul 20.1, modificând tipul funcţiei din **int** în **int***. Funcţia trebuie să introducă adresa obiectului găsit sau pointerul **NULL** dacă obiectul nu este în tablou.

20.5. TABLOURI DE POINTERI

Pointerii, fiind variabile, *pot forma alte tipuri de date structurate*, de exemplu tablouri. Dacă **T** este un tip de date oarecare, tipul pointerilor va fi **T***.

Un tablou de pointeri către tipul **T** se declară prin:

```
T     *x[5];
```

care se interpretează ca un tablou de 5 variabile, fiecare din acestea fiind pointer către tipul **T**. Elementele tabloului, **x[0]**, **x[1]**, etc. se pot prelucra acum ca variabile pointer. De exemplu :

```
char *a[3]={"abcd", "12345", NULL};
*a[1]++='a';
(*a[1])++;
```

sunt instrucţiuni corecte. a este un tablou de 3 pointeri, care sunt iniţializaţi cu adresele şirurilor constante „abcd" şi „1234", plasate în memorie de către compilator, respectiv cu pointerul 0 (NULL).

Dacă **a[1]** este un pointer, atunci * **a[1]** este primul caracter de la adresa **a[1]**, căruia i se atribuie 'a', după care se incrementează pointerul **a[1]**. În al doilea caz, se incrementează caracterul de la adresa **a[1]**. În urma acestor instrucţiuni, al doilea şir va fi „**a3345**". Să remarcăm că operatorii ++/-- „leagă" mai tare decât *, de aceea în primul caz nu sunt necesare paranteze.

Iată un alt **exemplu:** să presupunem că avem 10 şiruri de caractere, ale căror adrese se găsesc într-un tablou de pointeri (fig. 20.3) şi dorim o funcţie care să afişeze toate aceste şiruri, la un singur apel. Funcţia se poate construi astfel :

```
char *şir[10];
void write_lines (char *tp[], int n)
{
    int i;
    for (i=0; i<n; i++)
        if (tp[i]!=NULL)
            printf ("%s\n", tp[i]);
}
```

iar apelul corespunzător va fi:

```
write_lines(şir, 10);
```

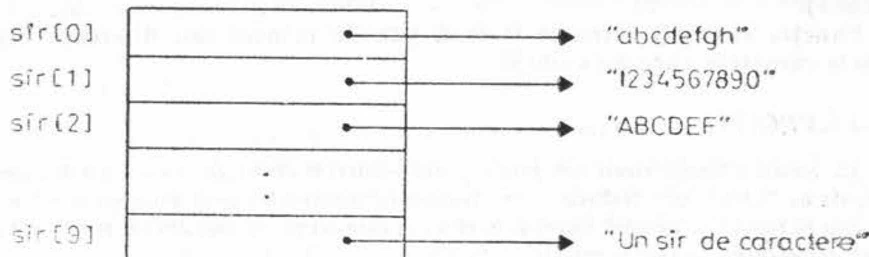


Fig. 20.3.

Specificatorul de format %s din **printf** corespunde cu **pointer către caracter**, care este chiar tipul lui **tp[i]**.

Un alt **exemplu important** este *reordonarea alfabetică a unor șiruri de caractere*; acest lucru se poate face comparând șirurile cu funcția de **bibliotecă strcmp()** și parcurgând tabloul până când este ordonat:

```
void sort_lines(char *tp[], int n)
{
        int sortat =0;
        char *temp;
        while (!sortat) {
              sortat=1;
              for (i=0; i<n-1; i++)
                    if (strcmp(tp[i], tp[i+1])>0) {
                          temp=tp[i+1];
                          tp[i]=tp[i+1];
                          tp[i+1]=temp;
                          sortat =0;
                    }
              }
        }
```

cu apelul corespunzător:

```
sort_lines (șir, 10);
```

Trebuie observat că se compară șirurile indicate de **tp[i]** și **tp[i+1]**, element cu element, dar de schimbat se schimbă pointerii din tabel, nu șirurile în sine (toate caracterele rămân în aceeași poziție în memorie) (fig. 20.4). De asemenea, elementele din **tp** se schimbă efectiv, deoarece **tp** este tablou, iar în funcție putem avea acces la elementele tabloului.

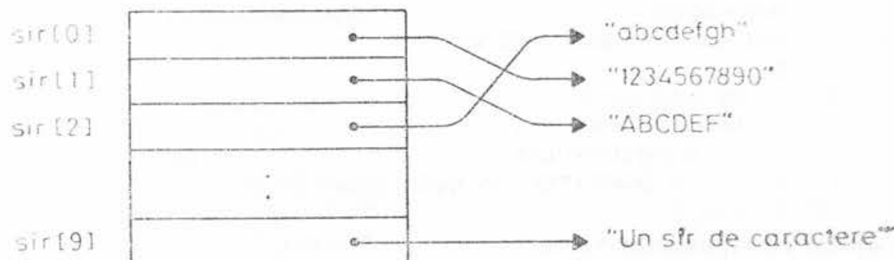


Fig. 20.4.

Se poate pune problema cum am putea citi cele 10 şiruri de caractere de la consolă, unde să le rezervăm spaţiu de memorie şi cum să iniţializăm tabloul de pointeri cu adresele acestor şiruri. Este important de observat că declaraţia tabloului **sir** rezervă spaţiu doar pentru cei 10 pointeri, nu şi pentru şirurile de caractere în sine.

O soluţie elegantă este folosirea funcţiei standard **strdup**, care are prototipul :

char *strdup(const char *s);

Această funcţie „salvează" şirul indicat de s într-o zonă de memorie dinamică (nu trebuie să ne preocupe unde anume) şi întoarce un pointer către zona respectivă sau NULL, dacă salvarea nu s-a putut face corect. Citirea şirurilor s-ar putea face acum cu o funcţie de forma:

```
void read_lines (char *tp[], int n)
{
    int i;
    char buf[80];
    for (i=0; i<n; i++)
        tp[i]=strdup (gets(buf));
}
```

iar apelul va fi :

```
read_lines (sir, 10);
```

Am rezervat explicit spaţiu doar pentru un şir de manevră **buf**. Apelul **gets(buf)** citeşte de la consolă un şir (până se întâlneşte un '\n'), punând caracterele citite în **buf**, inclusiv terminatorul '\0' şi întoarce **buf** (adică adresa lui **buf[0]**). Aceasta este transmisă funcţiei **strdup()**, care salvează şirul din **buf** „undeva", întorcând adresa zonei unde l-a salvat; în fine, această adresă este atribuită lui **tp[i]**.

Această tehnică permite citirea unui număr variabil de linii de la consolă. Să presupunem că, dacă se introduce '\n' pe o linie goală, introducerea trebuie să se termine. Dorim să citim cel mult 10 şiruri, dar să ne oprim la '\n' pe o linie goală. Beneficiem şi aici de funcţia **gets()**: aceasta *nu pune* '\n' în şir, dar pune totdeauna terminatorul '\0', aşa că dacă am introdus '\n' pe o linie goală, în **buf** vom găsi numai terminatorul (şirul vid). Funcţia **read_lines()** se poate scrie în acest caz :

```
void read_lines (char *tp[], int n)
{
    int i;
    char buf [80];
    for (i=0; i<n; i++) {
        gets (buf);
        if(*buf=='\0')
            break;
        tp[i]=strdup(buf);
    }
}
```

Dacă tabloul **şir[]** este în clasa **static** (declarat în exteriorul tuturor funcţiilor sau declarat explicit **static**), atunci el va fi iniţializat cu 0, adică cu pointerul **NULL**; astfel, dacă se fac mai puţine citiri, ceilalţi pointeri vor fi NULL-i,

ceea ce asigură o prelucrare corectă. Dacă operația are loc de mai multe ori, sau dacă **sir[]** este în clasa **auto**, trebuie avut grijă ca acesta să fie „șters" în prealabil (toți pointerii **sir[i]** să conțină NULL).

EXERCIȚII

14. Scrieți un program principal care să testeze funcțiile **read_lines**, **sort_lines** și **write_lines** de mai sus.

15. Modificați funcția **read_lines**, poziționând toți pointerii din tablou la valoarea NULL, înainte de a face citirile propriu-zise.

***16.** Scrieți o funcție de căutare liniară a unui șir dat de caractere într-un tablou de pointeri la caractere (tablou de șiruri). Funcția are ca parametri tabloul de pointeri, dimensiunea acestuia și pointerul la șirul de caractere care se caută. Funcția trebuie să întoarcă indicele șirului dat în tablou sau —1, dacă șirul dat nu există în tablou. Prototipul funcției este deci :

```
int caută_șir (char *tab[], int n, char *obiect);
```

Pentru comparația a două șiruri de caractere se poate folosi funcția de bibliotecă **strcmp**, care primește 2 pointeri către **char** și întoarce un **întreg** <0, 0 sau >0, după cum primul șir este mai mic, egal sau mai mare decît al doilea. Noțiunile de mai mare și mai mic se referă la relația de ordine alfabetică.

Scrieți un program principal în care definiți și inițializați un tablou de pointeri către **char** de forma :

```
char *tab_șir[5]={"12346790", "abcdef", "qwerty", "ABCDEF", "1993"};
```

și două șiruri constante de forma :

```
char *șir1="ABCDEF";
char *șir2="Acest șir nu va fi găsit";
```

apelând funcția de căutare cu cele două șiruri date, prin :

```
i=caută_șir(tab_șir, 5, șir1);
i=caută_șir (tab_șir, 5, șir2);
i=caută_șir (tab_șir, 5, "Un șir constant");
```

și interpretând corespunzător rezultatul căutării (deci valoarea lui i).

20.6. POINTERI CĂTRE POINTERI

Pointerii, fiind variabile, *ocupă spațiu de memorie* (dacă nu sunt în clasa **register**), deci *au asociate adrese de memorie*. Adresa unei variabile pointer va fi de tip **pointer către pointer**. Un **pointer către** un **pointer către** tipul **T** se declară prin :

```
T **p;
```

E de presupus că **p** va conține adresa unei variabile de tip **T ***, adică a unui **pointer** către **T**. Acționează evident operatorii & și *, de exemplu :

```
char *p, **q;
q= &p;
```

În această situație, **q** este pointer către pointer către **char**, ***q** este pointer către **char**, iar ****q** este primul **char** din șirul indicat de ***q** (*fig.* 20.5). Se vede că pointerii dubli oferă de fapt posibilitatea unor indirectări duble.

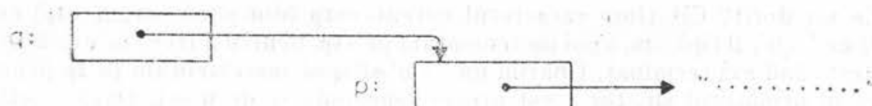


Fig. 20.5.

Tot ce este valabil la pointeri simpli, este valabil şi la pointeri dubli ; o funcţie care schimbă între ei doi **pointeri către un tip oarecare TIP** trebuie să fie de forma :

```
void swap(TIP **pp, TIP **pq)
{
    TIP *temp;
    temp=*pp; *pp=*pq ; *pq=temp ;
}
```

Similitudinile dintre tablouri 1-dimensionale şi pointeri ne arată acum că tablourile de pointeri pot fi accesate cu pointeri dubli. De asemenea, se menţin cele spuse la aritmetica pointerilor.

De **exemplu**, considerând declaraţiile :

```
char *a[5];
char **p;
```

se observă că **a** şi **p** sunt tablou, respectiv pointer către acelaşi tip de date, anume **char ***, deci putem scrie :

```
p=a;
```

Pointerul **p** va conţine adresa primului element al tabloului **a**, iar p+1, p+2, etc. vor indica pe a[1], a[2], etc. Faptul că elementele tabloului **a** sunt pointeri, nu schimbă cu nimic cele spuse la tablouri 1-dimensionale.

Funcţia **write_lines** din paragraful precedent se poate scrie acum astfel :

```
void write_lines (char **tp, int n)
{
    while (n--)
        printf("%s\n", *tp++);
}
```

Aici ***tp** este pointer simplu către **char**, deci corespunde cu formatul %s din **printf**, iar **tp++** va face ca să se treacă la următorul element al tabloului. Să observăm că se incrementează variabila locală **tp** din funcţie, ceea ce nu se va reflecta în variabila cu care este apelată această funcţie.

În anumite situaţii sunt necesare precauţii, anume atunci când poate apare un transfer prin referinţă, deşi nu aceasta a fost intenţia noastră. Să presupunem că, în funcţia **write_lines**, vrem să facem tipărirea nu cu **printf()**, ci la nivel de caracter, utilizând **putchar()** (exemplul este cu totul forţat!). **O soluţie posibilă**, dar cu totul **greşită**, este următoarea :

```
void write_lines (char **tp, int n)
{
    /*Varianta greşită!!*/
    while (n--) {
        while (**tp)
            putchar (*(*tp)++) ;
        putchar ('\n');
        tp++ ;
    }
}
```

Ce s-a dorit? Cît timp caracterul curent **tp (din şirul curent *tp) este diferit de '\0', îl tipărim, apoi incrementăm pe *tp, pentru a trece la următorul caracter; când am terminat, tipărim un '\n' şi apoi incrementăm pe tp pentru a trece la următorul şir, tot acest proces repetîndu-se de n ori. Dacă apelăm funcţia în contextul :

```
char *şir[]={"abc", „1234", „ABCDE"};
write_lines (şir, 3);
write_lines (şir, 3);
```

se va constata că la primul apel tipăririle se fac corect, în schimb, la al doilea, nu se mai tipăresc decît 3 '\n'-uri.

Eroarea este destul de ascunsă şi constă în faptul că, în urmă primului apel, şir[0], sir[1] şi sir[2], care iniţial indicau primele caractere din şirurile respective, sunt modificate, indicând acum terminatoarele '\0' din fiecare şir. Normal că, la al doilea apel, condiţia din **while(**tp)** este falsă de la început, şi nu se mai tipăreşte nimic. De ce se întâmplă această modificare? Deoarece incrementăm în funcţie pe *tp, adică un element al tabloului, iar ceea ce s-a transmis ca parametru actual a fost adresa acestui *tp, adică adresa primului element al tabloului. Astfel, a rezultat un transfer prin referinţă, deşi nu explicit, cu consecinţe dezastruoase (o funcţie care distruge date) (fig. 20.6).

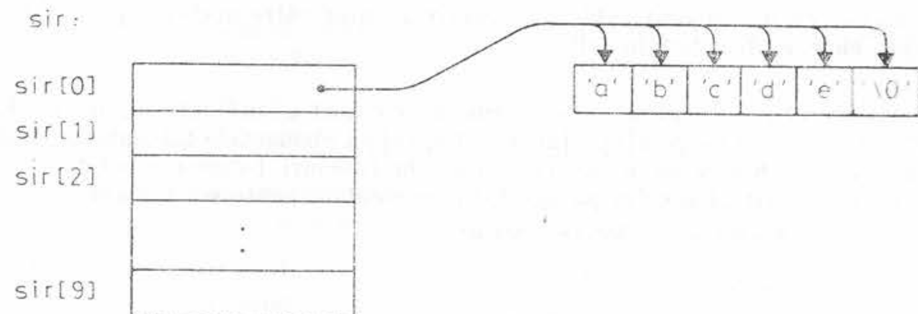


Fig. 20.6.

Soluţia corectă este folosirea unei variabile locale în funcţie, pentru accesul la elementele şirului curent, şi incrementarea acestei variabile locale (fig. 20.7):

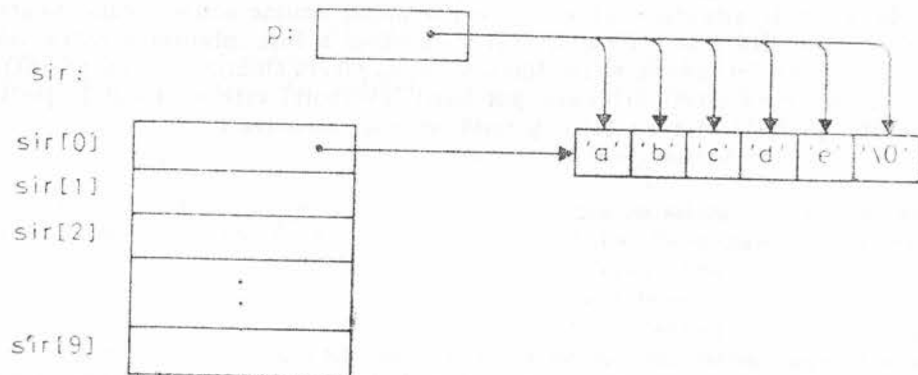


Fig. 20.7.

```
void write_lines (char **tp, int n)
{
    /*Varianta corectă!!*/

    char *p;
    while (n-- ) {
        p=*tp;
        while (*p)
            putchar (*p++);
        putchar ('\n');
        tp++;
    }
}
```

Pointerul local **p** este inițializat cu ***tp**, parcurgînd astfel șirul, iar de incrementat se incrementează acum **p**, ceea ce nu mai produce neplăceri. Să mai remarcăm că incrementarea lui **tp** nu este o eroare, deoarece **tp** este parametru formal al funcției și, ca atare, este transmis prin valoare, deci incrementarea lui nu se reflectă în afara funcției.

EXERCIȚII

17. Rescrieți funcțiile **read_lines** și **sort_lines**, înlocuind tabloul de pointeri cu un **pointer către pointer**. Verificați că nu se produce un transfer nedorit prin referință și că cele două funcții operează corect.

18. Scrieți un program principal care să pună în evidență aritmetica unui **pointer către char** și aritmetica unui **pointer către pointer către char**. Definiți doi asemenea pointeri inițializați corespunzător și apoi incrementați-i. Ce concluzie trageți despre dimensiunea unei adrese (număr de biți) din implementarea respectivă a limbajului C ?

20.7. POINTERI CĂTRE TABOLOURI CU O DIMENSIUNE

Acest tip de variabile se declară, de exemplu, prin :

int (*p)[10];

declarație care „spune" că **p** este un pointer către un tablou de 10 întregi. De observat parantezele: fără ele, declarația ar fi „spus" că **p** este un tablou de 10 pointeri către întregi, ceea ce este cu totul altceva. Ca regulă empirică de „citire" a unei asemenea declarații, putem considera: substituim formal ***p** cu un nume de variabilă, de exemplu **a**, declarația fiind deci **int a[10];**, deci **a** este un tablou de 10 întregi; atunci **p** este un **pointer către un tablou de 10 întregi.** Această regulă se poate aplica la orice declarație în care apar pointeri.

Care este utilitatea unor asemenea pointeri? În exemplul de mai sus, **p** va reprezenta adresa unui tablou de 10 întregi, dar am văzut că acest lucru e sinonim cu adresa primului element al tabloului, adică adresa unui întreg. Care este atunci diferența dintre un pointer către întreg și un pointer către un tablou de 10 întregi, căci, până la urmă, ei reprezintă fizic același lucru, anume adresa de început a tabloului?

Diferența apare în **aritmetica** acestor pointeri. Incrementarea lui **p** îl va deplasa peste 10 întregi, astfel ca el să indice, conform regulii generale de

la pointeri, **„următorul obiect din memorie, de același tip"**, adică următorul tablou de 10 întregi. Să considerăm următorul program:

```
#include <stdio.h>
int a[10];
void main ()
{
      int (*p_tab) [10], *p_int;
      p_tab= &a;
      p_int =a;
      printf ("p_int = %p p_tab= %p\n", p_int, p_tab);
      p_int++; p_tab++;
      printf ("p_int+1= %p p_tab+1= %p\n", p_int, p_tab);
}
```

Se atribuie lui **p_tab** „adresa tabloului a" (se putea scrie la fel de bine **p_tab=a,** deoarece am văzut că **a,** **&a** și **&a[0]** sunt sinonime; se preferă forma **p_tab=&a,** pentru a pune în evidență că **p_tab** este un pointer către tablou). Similar, se atribuie lui **p_int** adresa primului element al tabloului. La prima tipărire, cei doi pointeri vor avea aceeași valoare, dar la a doua, **p_int** va fi mai mare cu 2, iar **p_tab** cu 20, în ipoteza că întregii sunt pe 2 octeți (tipărirea cu format **%p** este specifică variabilelor pointer, și se face în hexazecimal). Similar, dacă se substituie peste tot tipul **int** cu tipul **long,** atunci diferențele vor fi de 4, respectiv 40 de octeți (fig. 20.8).

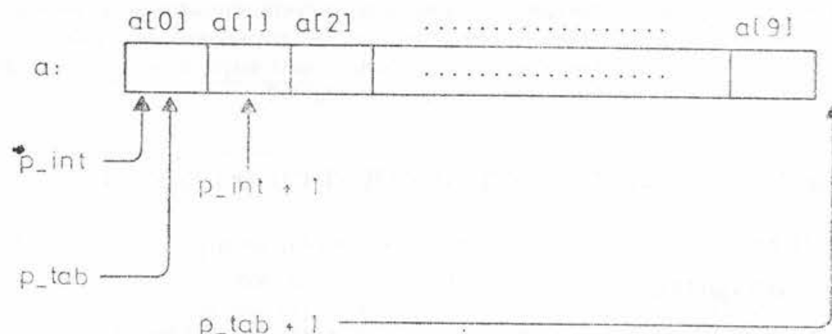


Fig. 20.8.

Deci, în esență, **pointerii către tablouri de tip T** au o aritmetică în care *unitatea este* **dimensiunea tabloului * sizeof(T),** spre deosebire de **pointerii către tipul T,** care au o aritmetică în care unitatea este **sizeof(T).**

Acest tip de date se folosește destul de rar, dar este esențial pentru a înțelege tratarea tablourilor cu mai multe dimensiuni în C.

20.8. POINTERI ȘI TABLOURI CU MAI MULTE DIMENSIUNI

Tablourile cu mai multe dimensiuni se declară prin:

T nume[d1][d2]...[dn];

unde T este tipul elementelor, iar **d1**, **d2**, ..., **dn** sunt constante întregi și pozitive. Iată câteva exemple:

```
int      a[3][5];
float    b[2][6][7];
```

Referirile la elementele tabloului se fac cu indici între 0 și dimensiunea respectivă −1, de exemplu b[0][0][0]...b[1][5][6] sunt referiri corecte la elementele tabloului **b**. Să observăm că fiecare indice se include între paranteze drepte, deci *se scrie corect* **a[i][j]** *și nu* **a[i,j]** ca în alte limbaje. Expresia **a[i,j]** este o expresie corectă sintactic, datorită operatorului virgulă, și înseamnă de fapt **a[j]**.

Prin definiție, elementele tabloului **nume** de mai sus, vor ocupa o **zonă continuă de memorie**, de dimensiune (în octeți):

d1×d2×... ×dn ×sizeof(T)

De exemplu, tabloul **b** va ocupa 84×4=336 de octeți, sau, mai corect spus, cât un tablou de 84 de **variabile de tip float**.

Cum sunt aranjate aceste variabile în memorie, adică, atunci când scriem **b[i][j][k]**, cum se calculează adresa acestui element, față de adresa primului element, adică b[0][0][0]?

Să considerăm declarația unui tablou **a**, cu tipul de bază **T** și dimensiuni (în ordine): d_1, d_2, ...d_n și să presupunem indicii:

i_1, i_2, ... ,i_n

indicii i_k luând valori cuprinse între:

$0 <= i_k <= d_k - 1$ cu k=1, 2, ..., n

Corespondența

(i_1, i_2, ..., i_n) −−−−−> adresa lui a[i_1][i_2]...[i_n]

se numește **funcția de alocare a memoriei** pentru tabloul **a**. Această funcție este identică pentru toate tablourile cu aceleași dimensiuni și același tip **T**. Notând cu d și i multiindicii:

$d=(d_1, d_2, ..., d_n)$

$i=(i_1, i_2, ..., i_n)$

și cu **T** tipul de bază al tabloului, putem nota funcția de mai sus prin:

$f_{d,T}(i, x)$=adresa elementului i al tabloului x, de tipul d și T (această funcție depinde de T și de multiindicele d).

În limbajul C, această funcție este:

$$f_{d,T}(i, x) = baza(x) + sizeof(T) * [\, i_1 d_2 d_3 \ldots d_n + i_2 d_3 d_4 \ldots d_n + i_3 d_4 d_5 \ldots d_n + \ldots + i_{n-1} d_n + i_n \,]$$

unde **baza(x)** este adresa de început a lui **x**, adică &**x[0][0]...[0]**

Această funcție se poate ține minte într-o formă sintetică, prin regula: „*tabloul este alocat la adrese succesive de memorie*, ultimul indice variind cel mai rapid". Considerând declarația:

```
char x[2][2][3];
```

imaginea în memorie a elementelor tabloului **x** va fi:

x[0][0][0]
x[0][0][1]
x[0][0][2]
x[0][1][0]
x[0][1][1]
x[0][1][2]
x[1][0][0]
x[1][0][1]
x[1][0][2]
x[1][1][0]
x[1][1][1]
x[1][1][2]

Adresa lui x[1][0][2] este:

baza (x)+1.2.3+0.3.2=baza(x)+8

elementul x[1][0][2] găsindu-se într-adevăr la distanţa de 8 octeţi de x[0][0][0].

În cazul tablourilor cu două dimensiuni (matrici), regula de alocare se mai poate ţine minte sub forma: *„matricile se memorează pe linii"*. Într-adevăr, un tablou x[2][3] va arăta în memorie astfel:

x[0][0]
x[0][1]
x[0][2]
x[1][0]
x[1][1]
x[1][2]

adică prima linie, apoi a doua.

Se observă faptul important că *funcţia de alocare nu depinde de prima dimensiune*. La tablourile 1-dimensionale, funcţia devine:

Adresa lui x[i]=baza(x)+sizeof(T)*i

Acest fapt ne permite ca, la declararea unui tablou 1-dimensional ca parametru al unei funcţii, să nu specificăm dimensiunea tabloului. Putem scrie de exemplu.

```
int strcmp(char a[ ]);
```

În limbajul C, *un tablou cu mai multe dimensiuni este tratat ca un tablou cu o dimensiune* (şi anume *prima*), care are ca elemente un tablou cu restul de dimensiuni; astfel, nu există limitări sintactice ale numărului de dimensiuni. De exemplu, tabloul:

```
int a[2][3];
```

este interpretat ca un tablou 1-dimensional cu 2 elemente, fiecare din elemente fiind un tablou 1-dimensional cu 3 elemente. Astfel, **a[0]** este un tablou de 3 întregi, şi la fel a[1].

Putem aplica acum ceea ce ştim de la relaţia dintre pointeri şi tablouri cu 1 dimensiune, şi de la **pointeri către tablouri.**

Astfel, dacă a[0] este un tablou de 3 întregi, el poate fi privit ca un **pointer constant** către un tablou de 3 întregi, şi la fel a[1]:

a[0]————>a[0][0] a[0][1] a[0][2]
a[1]————>a[1][0] a[1][1] a[1][2]

Elementele unui tablou, în cazul de față a[0] și a[1] se mai puteau scrie *a și *(a+1). Astfel, putem interpreta a[] ca un tablou de 2 pointeri, fiecare fiind **pointer către un tablou** de 3 întregi. Dar acum, un tablou de 2 pointeri poate fi văzut ca un **pointer către un pointer.** Pe scurt, dacă a este declarat de forma:

T a[d1][d2];

avem echivalările:

a[i][j]<==>(*(a+i))[j]<==>*(*(a+i)+j)

unde, la adunarea **a+i**, intervine **esențial** faptul că a este văzut ca un **pointer către un tablou de d2 obiecte de tip T,** deci ceea ce se adună este dimensiunea acelui tablou (adică a 2-a dimensiune a lui a), înmulțită cu **sizeof(T).**

Alte **expresii echivalente cu a[i][j]** sunt:

```
*(a[i]+j)            /* a[i]+j este pointer către tipul T         */
(*(a+i))[j]          /* a1+i este pointer către un tablou de d2   */
                     /* obiecte de tip T                          */
*(&a[0][0]+d2*i+j)   /* chiar funcția de alocare                  */
((T*)a)[d2*i+j]      /* se convertește a la T * și apoi se        */
                     /* simulează funcția de alocare              */
*(*(a+i)+j)          /* a este pointer către un tablou de d2      */
                     /* obiecte de tip T                          */
```

În diverse situații, se pot folosi diverse forme, toate având utilitatea lor.

Parcurgerea unui tablou cu mai multe dimensiuni se face de regulă printr-un *număr de cicluri* (egal cu numărul de dimensiuni), *incluse unul în altul, cu indici* care variază de la 0 la dimensiunea respectivă minus 1. **De exemplu,** tipărirea unei matrici se poate face cu secvența:

```
int a[10][10];
int i, j;
for (i=0; i<10; i++) {
        for (j=0; j<10; j+)
                printf("%6d", a[i][j]);
        printf ("\n");
}
```

EXERCIȚII

19. Scrieți un program principal în care este definit un tablou de întregi cu două dimensiuni. Parcurgeți elementele tabloului în două variante, pe linii și pe coloane, afișând la consolă adresa elementului curent. Verificați că adresele variază conform funcției de alocare a tabloului respectiv.

20. Verificați identitatea dintre expresiile echivalente cu a[i][j] în cazul unui tablou bidimensional, afișând la consolă toate aceste forme, pentru fiecare element al tabloului.

21. Se consideră un sistem de N puncte materiale, caracterizate prin coordonatele carteziene (în spațiu) ale fiecărui punct și prin masa asociată fiecărui punct. Definiți o structură adecvată de tablou pentru a reprezenta acest sistem de puncte materiale. Scrieți o secvență de program care să calculeze coordonatele centrului de greutate al sistemului de puncte materiale.

22. Scrieți un program principal în care să fie definită o matrice pătrată de numere reale de dimensiune 5. Cum se poate scrie o secvență care să realizeze transpunerea matricii ?

20.9. TRANSMITEREA UNUI TABLOU CU MAI MULTE DIMENSIUNI LA O FUNCȚIE

În toate situațiile în care lucrăm cu tablouri cu mai multe dimensiuni, trebuie să ținem cont de funcția de alocare a tabloului și să ne întrebăm: *„știe compilatorul să calculeze corect adresa unui element al tabloului?“*. Pentru aceasta, să reținem că e nevoie de toate dimensiunile tabloului, mai puțin prima.

Să considerăm o funcție care să tipărească elementele unei matrici de întregi. Ca la orice transmitere de tablou, ceea ce se transmite la funcție este **adresa de început**, dar aici trebuie să furnizăm și informațiile necesare pentru calculul adresei, în speță, dimensiunea a doua. Este **INCORECTĂ** forma:

```
void matprint (int a[ ] [ ], int m, int n)
{
    int i, j;
    for (i=0; i<m; i++) {
        for (j=0; j<n; j++)
            printf ("%6d", a[i][j]);
        printf ("\n");
    }
}
```

deoarece nu se poate calcula adresa lui a[i][j] (în momentul compilării nu este cunoscută a doua dimensiune a tabloului), deci apare eroare chiar la compilare. Dacă dorim tipărirea unei matrici declarate prin:

```
int x[3] [4];
```

atunci prima linie din definiția funcției trebuie să fie:

```
void matprint (int a[ ][4], int m, int n)
```

iar apelul va fi:

```
matprint (x, 3, 4);
```

Acum compilatorul știe să calculeze adresa lui a[i][j], dar această funcție va putea fi folosită numai cu matrici cu 4 coloane, ceea ce este inacceptabil. Ceea ce se urmărește este scrierea unei funcții care să tipărească corect orice matrice. **Soluția corectă** este să folosim una din formele echivalente pentru a[i][j], anume aceea care simulează funcția de alocare, pe baza celei de-a doua dimensiuni, adică pe baza lui n:

```
void matprint (int (*a) [ ], int m, int n)
{
    int i, j;
    for (i=0; i<m; i++) {
        for (j=0; j<n; j++)
            printf ("%6d", ((int *) a) [i*n+j]);
        printf ("\n");
    }
}
```

Ceea ce se transmite este adresa de început a tabloului **a**, văzută de tip **int (*)[]** (adică **pointer către un tablou**), în felul acesta nemaiavând erori la compilare. O variantă mai clară este declararea parametrului a de tip **void ***.

Conversia explicită de tip **((int *)a)** face ca **a** să fie văzut ca un pointer către **int**, deci ca un tablou 1-dimensional. Indicele i*n+j face ca să fie accesat corect elementul a[i][j] al matricii, indiferent de dimensiuni. O soluţie mai explicită ar fi declararea şi iniţializarea separată a unui pointer de tip **int** *:

```
int *v=(int *) a;
```

şi folosirea lui în **printf**:

```
printf("%6d", v[i*n+j]);
```

O variantă „exclusiv cu pointeri" a acestei funcţii este:

```
#include <stdio.h>
#include <stddef.h>
#define FORMAT "%6d "
void mat_print (void *a, int m, int n)
{
    int *v=(int *) a;
    int mn=m*n;
    ptrdiff_t d;
    while ((d=v—(int *)a)< mn)
         printf ((d%n==n—1)?FORMAT"\n": FORMAT, *v++);
}
```

Se calculează numărul *mn* de elemente ale matricii şi se declară un pointer **v** de tip **int** *, iniţializat cu adresa de început a matricii. Cât timp diferenţa dintre acest pointer şi adresa de început a matricii este mai mică decât *mn*, se tipăreşte elementul curent şi apoi se incrementează pointerul **v**. Tipul de date predefinit **ptrdiff_t** este adecvat diferenţei a 2 pointeri. Afişarea caracterului '\n' are loc după fiecare al n-lea element, deci după afişarea unei linii. Se observă expresia condiţională prin care se selectează un format sau altul de afişare. Expresia FORMAT„\n" înseamnă concatenarea şirurilor constante FORMAT şi „\n".

Multe din operaţiile uzuale cu matrici se implementează mult mai eficient prin operaţii cu pointeri. Iată o funcţie care întoarce urma unei matrici pătrate (suma elementelor de pe diagonala principală):

```
int trace (void *a, int n)
{
    int *v=(int *)a;
    int tr=0;
    int m=n++;
    while (m——) {
        tr+=*v;
        v+=n;
    }
    return tr;
}
```

Dacă v este iniţializat cu adresa de început a matricii, se observă că adunarea cu n+1 face ca v să parcurgă diagonala principală. Elementele de la adresa v se sumează în variabila tr, care este întoarsă programului ape-

lant. Operația se realizează prin 3*n adunări, fără nici o înmulțire. Varianta „standard" a acestei funcții ar fi fost:

```
int trace (void *a, int n)
{
    int *v=(int *)a;
    int tr=0;
    int i;
    for (i=0; i<n; i++)
        tr+=v[i*n+i];
    return tr;
}
```

în care se fac 3*n adunări și n înmulțiri, ceea ce, dacă n este mare, contează.

În mod evident, operațiile cu tablouri cu mai mult de 2 dimensiuni se fac după modelul de la tablouri cu 2 dimensiuni.

20.10. CALIFICATORUL CONST APLICAT LA POINTERI

Calificatorul **const** se poate aplica la variabile inițializate, orice încercare ulterioară de modificare a variabilei fiind semnalată ca eroare de compilare. De exemplu, situația:

```
const char x='a';
x++;
```

va fi semnalată ca eroare.

Putem folosi **const** și la pointeri, în sensul că putem declara **pointeri constanți** sau **pointeri către constante.**

Restricțiile impuse diferă de la caz la caz și sunt lămurite de **exemplul** următor:

```
const   int a=10, *pc= &a;
const   int *const cpc=pc;
int     b, *const cp= &b;
```

Aici a este un întreg constant (a nu se poate modifica), pc este un pointer către un întreg constant (pc se poate modifica dar *pc nu), cpc este un **pointer constant către un întreg constant** (nici cpc nici *cpc nu se pot modifica), iar cp este un **pointer constant** (**cp** nu se poate modifica dar ***cp** da). Instrucțiunile următoare sunt **ilegale**:

```
a=1; a++; *pc=2; cp= &a; cpc++;
```

în timp ce instrucțiunile următoare sunt **corecte**:

```
b=a; *cp=a; pc++; pc=cpc;
```

EXERCIȚII

23. Verificați că se obțin erori de compilare în cazul operațiilor ilegale descrise mai sus.

20.11. ARGUMENTE ÎN LINIE DE COMANDĂ

O aplicație importantă a tablourilor de pointeri este transmiterea argumentelor din linia de comandă către funcția **main,** într-un mod independent de implementarea limbajului.

Când se lansează un program executabil de la consolă (de exemplu, în MS-DOS, de tip .exe sau .com), se pot introduce și argumente (parametri) în linia de comandă.:

```
>xcopy a\f1.c c\f2.c
```

Numele programului este **xcopy**, iar șirurile de caractere „a\f1.c" și „c\f2.c"constituie argumentele programului.

În C, putem face ca funcția **main** să aibă acces la aceste argumente. Pentru aceasta, se declară **main** ca având 2 parametri, unul de tip întreg, și celălalt un tablou de pointeri la **char**, de dimensiune neprecizată. Uzual, acești parametri sunt „botezați" **argc** (de la argument count) și **argv** (de la argument values), dar se pot folosi orice nume. Primul parametru spune numărul de argumente din linia de comandă și este cel puțin 1, iar al doilea conține pointeri (în număr egal cu primul parametru) către șirurile de caractere care constituie argumentele propriu-zise. Primul șir este totdeauna numele sub care este cunoscut programul de către sistemul de operare. În urma declarației:

```
void main (int n, char *s[ ])
```

s[0] va indica numele programului (de fapt, numele complet al fișierului care conține programul executabil), iar s[1], s[2], ..., s[n−1] vor indica șirurile respective (dacă n este >=2). Standardul ANSI garantează ca s[n] este pointerul NULL. Astfel, putem avea acces la argumente.

Exemplul clasic în acest context este un *program care își tipărește la consolă toate argumentele, inclusiv numele propriu:*

```
void main (int n, char *s[ ])
{
    int i;
    for (i=0; i<n; i++)
            printf ("%s", s[i]);
    printf ("\n");
}
```

O altă variantă este să declarăm al doilea parametru al lui **main** de tip **char** **:

```
void main(int argc, char **argv)
{
    while (argc--)
            printf ("%s", *argv++);
    printf ("\n");
}
```

Aici, **argv** este de tip **char** **, deci ***argv** este de tip **char** *, corespunzând cu formatul %s; după acces, se incrementează **argv**, indicându-se astfel pointerul următor.

În implementările care folosesc standardul ANSI, mai este posibilă o variantă, care se bazează pe faptul că **argv[argc]** este obligatoriu NULL, dar această variantă nu este recomandabilă, pentru că este posibil să funcționeze pe implementări mai vechi ale limbajului:

```
void main (int argc, char **argv)
{
    while (*argv!=NULL)
        printf ("%s", *argv++);
    printf ("\n");
}
```

În standardul ANSI se mai introduce un al treilea parametru (opțional) al lui **main**, tot de tip tablou de pointeri la **char**, care conține o descriere a „mediului" în care este rulat programul (environment). Ce conțin aceste șiruri depinde de implementare (de exemplu, sub MS-DOS, sunt transmise PATH-urile active în momentul execuției, numele interpretorului curent de comenzi etc.). Aceste informații pot fi utile la execuția unor alte programe (procese), când se transmite environment-ul (identic sau modificat) către acestea. Tehnica de acces este aceeași (ultimul pointer este garantat a fi NULL):

```
void main (int argc, char **argv, char **env)
{
    printf ("Argumente:");
    while (argc--)
        printf ("%s", *argv++);
    printf ("\nMediu");
    while (*env!=NULL)
        printf ("%s", *env++);
    printf ("\n");
}
```

Faptul că al 3-lea parametru (**env**) poate lipsi în **main** provine din modul de transfer al parametrului la o funcție: transmiterea are loc prin stivă, ordinea în care sunt puși parametrii în stivă este de la dreapta la stânga, iar stiva este descărcată de programul apelant. Astfel, în vârful stivei se vor găsi, în ordine: **argc**, **argv** și **env**. Faptul că **main** este declarată cu sau fără parametrul **env** nu are nici o influență asupra stivei la execuție. Din același motiv putem declara **main** fără parametri: cei 3 parametri există în stivă, dar nu vor fi folosiți.

EXERCIȚII

24. Scrieți un program care să-și afișeze toate argumentele și environment-ul. Lansați acest program în execuție printr-o linie de comandă în care, după numele programului, introduceți diverse șiruri de caractere separate prin blank. Observați forma în care apare numele programului (dependentă de sistemul de operare folosit).

25. Scrieți un program care să realizeze copierea unui fișier disk în alt fișier disk, numele celor două fișiere fiind preluate din linia de comandă. Afișați un mesaj de eroare dacă nu s-au introdus două argumente în linia de comandă. Pentru operații cu fișiere se pot folosi funcțiile de bibliotecă **fopen**, **fclose**, **fgetc**, **fputc**. Studiați aceste funcții din meniul de HELP pe care-l aveți la dispoziție.

20.12. FUNCȚII CARE ÎNTORC POINTERI

Funcțiile pot întoarce orice fel de variabile simple deci și pointeri către diverse tipuri de date. Trebuie acordată atenție întoarcerii de către funcție a adresei unei variabile locale funcției: obligatoriu, această variabilă trebuie să fie în clasa **static**. Cu alte cuvinte, *nu avem voie să folosim adresele unor variabile din clasa* **auto** *în afara funcției în care ele sunt definite*, deoarece aceste

variabile dispar din memorie la încheierea execuției funcției. Concret, această dispariție are loc prin descărcarea zonei de stivă (și folosirea ei în alte scopuri), la ieșirea din funcție. Mai rezultă, de asemenea, că *în nici un fel nu trebuie întoarsă o adresă a unui parametru actual al funcției*; aceștia sunt transmiși prin stivă întotdeauna și, pe durata execuției funcției, rămân în stivă. La revenirea în programul principal, stiva este descărcată și zonele de stivă respective vor căpăta alte destinații. Deci **nu trebuie scris niciodată** ceva de genul:

```
int *f(void)                    int *g (int i)
{                               {
    int i;                          . . . . .
    . . . . . . .                   . . . . .
    return &i;                      return &i
}                               }
```

Ambele situații constituie **erori grave**, cu atât mai mult cu cât cele două funcții sunt corecte sintactic, deci nu vor apare erori la compilare.

Funcțiile pot întoarce pointeri către diverse tipuri, ca o acțiune **adiacentă** acțiunii lor de bază, ajutând scrierea mai comodă și mai compactă a programului.

De **exemplu**, funcția de bibliotecă **strcpy()** întoarce adresa șirului destinație, care este totodată parametru formal. Ea se poate scrie în forma:

```
char *strcpy (char *dest, const char *sursa)
{
    char *p=dest;
    while (*p++=*sursa++)
        ;
    return dest;
}
```

Această formă permite acum acțiuni multiple, de exemplu copierea unui șir și calculul lungimii șirului copiat într-o singură linie de program:

```
n=strlen(strcpy(a, b));
```

Utilitatea principală a funcțiilor care întorc pointeri este alocarea de spațiu pentru variabile dinamice. Acestei categorii de variabile nu i se alocă spațiu la compilare, ci la execuție. Ele nu au nume asociate și sunt accesate prin pointeri poziționați corespunzători. O funcție care alocă spațiu în mod dinamic pentru un obiect (variabilă), întorcând un pointer la spațiul rezervat (eventual inițializat cu anumite date precizate prin argumentele funcției), poartă numele de **constructor.** O funcție care primește un pointer la o zonă alocată dinamic și eliberează această zonă, se numește **destructor.**

Biblioteca standard pune la dispoziție 3 funcții care pot fi folosite în acest scop, având prototipurile:

```
void *malloc (size_t size);
void *calloc (size_t n, size_t size);
void free (void *p);
```

Apelul **malloc(size)** alocă spațiu continuu pentru un obiect având dimensiunea **size** octeți și întoarce un pointer către începutul acestui spațiu, sau pointerul NULL, dacă alocarea nu s-a putut face corect. Acest pointer poate fi convertit la orice tip de pointer.

Apelul **calloc(n, size)** alocă spațiu continuu pentru un tablou de n obiecte, fiecare presupus a avea **size** octeți, întorcând un pointer la spațiul rezervat sau NULL, dacă alocarea nu s-a putut face corect.

Apelul **free(p)** eliberează spațiul de memorie indicat de pointerul **p**. Este obligatoriu ca acest spațiu să fi fost alocat anterior cu **malloc** sau **calloc**.

Un **exemplu tipic** de folosire a lui **malloc** îl constituie funcția de bibliotecă **strdup**, care salvează un șir într-o zonă creată dinamic și întoarce un pointer de tip **char*** la acea zonă sau NULL, dacă nu s-a putut aloca spațiu. Ea poate fi scrisă astfel:

```
char *strdup (const char *s)
{
    char *p=(char *) malloc (1+strlen(s));
    if(p!=NULL)
        strcpy(p, s);
    return p;
}
```

Se rezervă spațiu cu **malloc** pentru a se memora șirul s (inclusiv terminatorul '\0') și, dacă pointerul întors este diferit de NULL, se copiază șirul s în zona respectivă, întorcând programului apelant adresa acestei zone. Un **exemplu** de apel este:

```
char *ptr;
ptr=strdup(str);
```

O situație deosebită o reprezintă funcțiile care întorc pointeri către zone alocate static în memorie.

Iată o funcție (**exemplu clasic**) care întoarce numele lunii calendaristice, primind numărul acesteia (între 1 și 12). Este eficient să se țină un tabel static de pointeri la șiruri constante de caractere, întorcând un asemenea pointer:

```
char *nume_luna(int n)
{
    static char *nume [ ]={
        "Ilegal",
        "Ianuarie", "Februarie", "Martie", "Aprilie"
        "Mai", "Iunie", "Iulie", "August", "Septembrie",
        "Octombrie", "Noiembrie", "Decembrie"
    };
    return (n<1 | | n>12)? nume[0]: nume[n];
}
```

La compilare se rezervă spațiu pentru șirurile constante dintre acolade, iar pointerii din tabloul **nume** sunt inițializați cu adresele acestor șiruri. Fiind declarat static, tabloul **nume** este inițializat o singură dată, iar toate apelurile vor întoarce aceleași adrese.

Evident că există pericolul ca o funcție care primește un asemenea pointer către o zonă statică să modifice această zonă din greșeală. În această situație, toate apelurile ulterioare ale funcției nume_luna vor întoarce pointeri către aceste zone distruse. Protecția împotriva unei situații de genul acesta

este renunțarea la date statice globale și întoarcerea către funcția apelantă, de fiecare dată, a unei alte zone de memorie, obținută prin alocarea dinamică:

```
char *nume_luna(int n)
{
    static char *nume [ ]={
        "Ilegal",
        "Ianuarie", "Februarie", "Martie", "Aprilie"
        "Mai", "Iunie", "Iulie", "August", "Septembrie",
        "Octombrie", "Noiembrie", "Decembrie"
    };
    char *p;
    n=(n>0 && n<13)? n:0;
    p=(char *) malloc (1+strlen(nume[n]));
    if(p!=NULL)
            strcpy(p, nume[n]);
    return p;
}
```

EXERCIȚII

*26. Scrieți un program principal care să citească de la consolă o matrice pătrată de numere reale. Dimensiunea matricii nu este cunoscută și trebuie citită de la consolă, înainte de citirea elementelor. Deoarece nu se cunoaște apriori dimensiunea tabloului, trebuie rezervat spațiu la execuție, prin **malloc** sau **calloc**. Scrieți o secvență de program care să afișeze pe linii matricea introdusă și apoi să elibereze zona de memorie alocată. Pentru accesul la elementele matricei (atât la citire cât și la scriere) trebuie folosită una din formele echivalente de simulare a funcției de alocare a tabloului.

27. Scrieți o funcție similară cu **calloc**, care să rezerve spațiu pentru un tablou de obiecte de un tip oarecare, întorcând același lucru ca și **calloc**. Pentru rezervare de spațiu, folosiți **malloc**.

20.13. POINTERI CĂTRE FUNCȚII

O funcție are asociată o adresă fixă de memorie, anume adresa de început a funcției. Un pointer către funcție va conține o asemenea adresă. Când se declară un pointer către o funcție, trebuie precizate toate informațiile despre funcție (tip, număr și tip parametri), deci ca și când s-ar declara o funcție de același tip și cu aceiași parametri. Aceste informații sunt necesare pentru a se putea apela indirect o anumită funcție, prin intermediul pointerului (trebuie știut câți parametri și de ce tip se transmit, ce tip se întoarce etc.).

Forma generală a declarației unui pointer la un tip de funcție este:

```
tip (*pf) (lista de parametri formali);
```

Se remarcă prezența parantezelor: fără ele declarația ar spune că **pf** este o funcție cu tipul **tip** * ceea ce este cu totul altceva. Ca regulă practică, declarația se face ca și cum am scrie un prototip de funcție dar numele funcției se substituie cu expresia **(*nume_pointer)**. De exemplu, declarația:

```
char *(*p) (char *, const char *);
```

spune că **p** este un pointer la o funcție de tip **char** *, care are doi parametri, unul de tip **char** * și celălalt de tip **const char** *. Ca și la prototipuri, nu este

neapărat necesar ca parametrii să fie denumiți: este suficientă precizarea tipului lor. Acum **p** poate primi ca valoare adresa de început a unei funcții de exact același tip și cu aceiași parametri.

În limbajul C, *numele unei funcții este sinonim cu adresa ei de început* (la fel ca la tablouri), deci pointerul definit mai sus ar putea primi ca valoare:

```
p=strcpy;
```

prin aceasta atribuindu-se lui p adresa de început a funcției **strcpy**.

Același lucru se obține prin:

```
p= &strcpy;
```

Dacă tipul pointerului nu coincide cu tipul funcției, se va semnala eroare la compilare.

Apelul unei funcții prin intermediul unui pointer se scrie:

```
(*pf)(lista de parametri actuali);
```

pentru funcții fără tip și, respectiv:

```
var=(*pf)(lista de parametri actuali);
```

pentru funcții cu tip.

Standardul ANSI face și identificarea reciprocă pointeri — funcții, permițând ca să nu se mai scrie operatorul de **indirectare** *, deci se pot scrie apeluri de forma:

```
pf(lista de parametri actuali);
var= pf(lista de parametri actuali);
```

O primă aplicație a acestor pointeri este construirea unor tablouri cu adrese de funcții, care pot fi apelate apoi indirect.

Să definim **de exemplu** un tablou de 6 pointeri la funcție de tipul **double**, care are un argument de **tip double** și să inițializăm acest tablou cu adresele unor funcții matematice de bibliotecă. Vom tabela valorile acestor funcții, cu valori de la 0.01 la 1.0, cu pasul 0.01:

```
#include <stdio.h>
#include <math.h>
double (*tab[ ]) (double)={sin, cos, tan, exp, log, log 10};
void main (void)
{
    int i;
    double x;
    for(x=0.01; x<1.01; x+=0.01) {
        for (i=0; i<sizeof(tab)/sizeof(tab[0]); i++)
                printf("%lf", (*tab[i])(x));
    printf ("\n");
}
```

Aici **tab[i]** este un element al tabloului, deci un pointer la funcție, ***tab[i]** este funcția, iar **(*tab[i]) (x)** este apelul funcției cu argumentul x, care întoarce o valoare **double**. Expresia **sizeof(tab) /sizeof(tab[0])** dă dimensiunea tabloului indiferent de tipul său, deci dacă se mai adaugă o funcție în lista de inițializare, ciclul după i nu trebuie modificat.

A doua **utilitate importantă a pointerilor la funcții** este *folosirea funcțiilor polimorfice*. Aceste funcții sunt proiectate pentru a se putea aplica oricăror tipuri de date. Evident ca prelucrările concrete ale datelor vor diferi

substanțial de la un tip la altul, dar aceste prelucrări se pot concentra în funcții specifice de prelucrare. Transmițând funcției polimorfice un pointer la funcția de prelucrare și utilizând peste tot tipul pointer universal **void** *, se poate realiza relativ ușor o asemenea funcție polimorfică.

De exemplu, funcția standard de bibliotecă **qsort** implementează algoritmul de sortare internă **Quicksort** (un *algoritm foarte performant*). Funcția de bibliotecă are prototipul:

```
void qsort(void *tab, size_t n, size_t dim,
            int (*cmp) (const void *, const void *));
```

Se sortează tabloul tab, având dimensiunea n, un element al tabloului având dimensiunea dim, iar cmp este un pointer la o funcție care primește doi pointeri la două elemente ale tabloului, întorcând o valoare negativă dacă primul element este mai mic decât al doilea, 0 dacă elementele sunt egale și o valoare pozitivă dacă al doilea este mai mic decât primul. Funcția **cmp** trebuie scrisă de utilizator. Sensul relațiilor mai mic, mai mare, respectiv egal, este abstract și este definit de utilizator (de exemplu, se pot sorta înregistrări în care cheia de comparație este doar o componentă a înregistrării etc.). În urma execuției, elementele tabloului vor fi în ordine nedescrescătoare, în sensul relației de ordine definite de funcția **cmp**.

Iată trei exemple de apel al acestei funcții.

a) **Primul exemplu** *sortează un tablou de întregi, de dimensiune 100.* Funcția de comparație trebuie să fie:

```
int cmp_num (const void *a, const void *b)
{
    return *((int *) a) — *((int *) b);
}
```

Se convertesc pointerii **(void *)** la **(int *)**, se ia apoi conținutul lor (deci se iau doi întregi) și se întoarce diferența celor doi întregi. Apelul va fi în acest caz:

```
int tab_num[100];
qsort (tab_num, 100, sizeof (int), cmp_num);
```

b) **Al doilea exemplu** își propune să *sorteze un tablou de pointeri la* **char**, *sortarea făcându-se în ordinea alfabetică a șirurilor* indicate de elementele tabloului. Pentru compararea propriu-zisă a două șiruri, se va folosi funcția de bibliotecă **strcmp**, care trebuie să primească doi pointeri la **char**. Funcția de comparație va fi atunci:

```
int cmp_pstr(const void *a, const void *b)
{
    return strcmp (*((char **) a), *((char **) b) );
}
```

Aici a și b sunt pointeri la elementele tabloului și, deoarece tabloul este de pointeri la **char**, tipul lui a și b este **char** **. Se convertesc a și b de la **(void *)** la **(char **)** și se ia conținutul acestor pointeri, care reprezintă acum variabile de tip **(char *)**. Aceste variabile se transmit la funcția **strcmp**. Apelul lui **qsort** este în acest caz:

```
char *tab_str[100];
qsort(tab_str, 100, sizeof(char *), cmp_pstr);
```

Trebuie observat că, în acest caz, șirurile de caractere nu-și modifică poziția din memorie. Ceea ce se schimbă sunt elementele tabloului de pointeri, care vor indica șirurile în ordinea alfabetică.

c) **Al treilea exemplu** *sortează un tablou de 100 de șiruri de caractere, fiecare de lungime maximă de 80 de octeți*, declarat prin:

```
char a[100] [80];
```

De data aceasta se dorește să se modifice efectiv poziția din memorie a celor 100 de șiruri, lucru care este realizabil, deoarece toate șirurile au același spațiu rezervat. Funcția de comparație se scrie în acest caz în forma:

```
int cmp_str (const void *a, const void *b)
{
    return strcmp (a, b);
}.
```

Ceea ce primește acum funcția de comparație sunt chiar adresele șirurilor, care trebuie comparate cu **strcmp**, deci acestea se transmit ca atare. Nu sunt necesare aici conversii explicite de tip, deoarece nu se face nici o operație de indirectare asupra lui a și b. Apelul este:

```
qsort(a, 100, 80, cmp_str);
```

ceea ce pune în evidență că un element al tabloului este în acest caz de dimensiune 80 de octeți.

EXERCIȚII

*28. Scrieți o funcție cât mai generală, care să schimbe între ele două obiecte oarecare (de aceeași dimensiune) din memorie. Funcția primește adresele celor două obiecte (de tip **void***) și dimensiunea în octeți a obiectelor. Testați această funcție pentru a schimba doi întregi, două numere reale etc. Pentru a obține dimensiunea unui anumit tip de date, se folosește operatorul **sizeof**.

29. Testați funcția de bibliotecă **qsort**, în diverse contexte: tablou de întregi, de float, de pointeri la șiruri.

30. Scrieți un program în care definiți mai multe funcții reale de o variabilă reală. Definiți un tablou de pointeri la acest tip de funcție, inițializat cu adresele funcțiilor respective. Scrieți o secvență de program care să calculeze integralele definite ale acestor funcții, pe un interval precizat și să afișeze la consolă integralele respective. Pentru calculul aproximativ al integralelor, folosiți metoda dreptunghiurilor. Definiți în prealabil o funcție C, care să calculeze o asemenea integrală, primind ca parametri un pointer la funcția matematică respectivă și cele două capete ale intervalului pe care se calculează integrala.

31. Rescrieți programul de la 20.30, înlocuind metoda dreptunghiurilor cu metoda trapezelor.

20.14. DECLARAȚIA TYPEDEF

Declarația typedef permite *redenumirea unor tipuri de date de către utilizator*. Este foarte eficientă la clarificarea declarațiilor complicate. Forma generală este similară unei declarații de variabilă de un anumit tip, în plus apărând cuvântul-cheie **typedef** pe prima poziție. În urma declarației, ceea ce era pe post de numele variabilei devine numele noului tip de date, care se poate folosi ulterior. Tradiția cere ca tipurile astfel definite să fie scrise cu majuscule. Iată câteva exemple:

```
typedef char *STRING;
typedef char BUFFER[80];
typedef double (*DPFD) (double);
typedef int (*PFCMP) (const void *, const void *);
```

Dacă ignorăm cuvântul-cheie **typedef**, atunci STRING ar fi o variabilă de tip **char** *, BUFFER ar fi un tablou de 80 de caractere, DPFD un pointer la o funcție de tip **double**, care are un argument de tip **double**, iar PFCMP un pointer la o funcție de tip **int**, care are două argumente, ambele de tip **const void** *. Atunci, noile tipuri rebotezate vor fi chiar tipurile de mai sus. Putem scrie acum declarații de forma:

```
STRING   s1, s2;
BUFFER   buf;
DPFD     tab[6];
PFCMP    p;
```

care spun foarte clar că s1 și s2 sunt variabile de tip STRING (deci **char** *), că **buf** este o variabilă de tip BUFFER (deci tablou de 80 de char), că **tab** este un tablou de 6 variabile de tip DPFD, (deci tablou de 6 pointeri la funcție de tip **double** cu un argument **double**) și, în fine, că p este o variabilă de tip PFCMP (deci un pointer la funcție de tip **int** cu 2 argumente **const void** *). Să comparăm simplitatea ultimelor două declarații cu cele ale declarațiilor echivalente fără **typedef**:

```
double (*tab[6]) (double);
int (*p) (const void *, const void*);
```

Odată definit un tip de date cu **typedef**, cu el se pot crea orice fel de variabile, deci se lucrează ca și cu un tip predefinit.

De exemplu, declarația:

```
DPFD    *p;
```

spune că p este un pointer către tipul DPFD, adică pointer către pointer la funcție de tip **double** cu un argument **double**, iar declarația:

```
BUFFER *f(void);
```

spune că f este o funcție fără argumente, care întoarce un pointer la tipul BUFFER, ceea ce e mult mai clar decât:

```
char (*f(void))[80];
```

Prototipul funcției de bibliotecă **qsort** se poate scrie acum:

```
void qsort (void *tab, size_t n, size_t dim, PFCMP cmp);
```

Folosirea disciplinată și gradată a lui **typedef** permite scrierea comodă și înțelegerea rapidă a unor declarații care altfel ar fi complet ilizibile.

De exemplu, declarația:

```
DPFD    *(*p) [3];
```

spune că p este un pointer la un tablou de 3 pointeri la tipul DPFD, dar putem defini acum:

```
typedef DPFD *(*CRAZY) [3];
```

care introduce noul tip CRAZY, adică poiner la un tablou de 3 pointeri la tipul DPFD. Declarația de funcție:

```
CRAZY *f(DPFD, DPFD);
```

poate fi încă înțeleasă: f este o funcție care are 2 argumente de tip DPFD și întoarce un pointer la CRAZY. Putem acum explica pas cu pas tipurile DPFD și CRAZY, obținând în final o declarație bazată numai pe tipuri simple.

Ca exercițiu, se poate încerca scrierea prototipului ultimei funcții f fără a folosi declarațiile **typedef** de mai sus.

Să vedem acum care este tehnica de scriere a unei **funcţii polimorfice**, dezvoltând o funcţie de căutare binară într-un tablou ordonat de obiecte distincte. Această funcţie este o *generalizare a funcţiei de căutare binară într-un tablou de întregi*, definită în 20.2.

Trecerea la funcţia polimorfică presupune întâi stabilirea tipurilor de date. Pentru generalitate maximă, trebuie folosit tipul **void** $*$, care este universal. Trebuie deci transmise adresa tabloului şi adresa cheii după care are loc căutarea, ambele de tip **void** $*$. În general cheia nu coincide ca tip cu un element al tabloului (poate fi de exemplu un câmp al unui element). Pentru calculul adreselor elementelor din tablou, e obligatorie cunoaşterea dimensiunii (în octeţi) a unui element. De asemenea, trebuie furnizat funcţiei un pointer la o funcţie de comparaţie, de tipul:

```
typedef int (*PFCMP) (const void *cheie, const void *elem);
```

unde **cheie** este adresa cheii de căutare (care nu este obligatoriu de acelaşi tip cu adresa unui element al tabloului), iar **elem** este adresa unui element al tabloului. Funcţia de comparaţie trebuie să întoarcă o valoare negativă, 0 sau pozitivă, după cum $*$ **cheie** este $\langle$, $=$ sau $\rangle$ decât $*$ **elem**, în sensul relaţiei de ordine definită de funcţia de comparaţie.

Funcţia bin_1 întoarce indicele elementului găsit sau —1. Într-o altă variantă, s-ar putea întoarce adresa obiectului găsit (de tip **void** $*$) sau NULL. Se vor ilustra ambele variante.

```
typedef unsigned char BYTE;
int bin_1(void *key, void *baza, size_t n, size_t size, PFCMP cmp)
{
    int cond, stinga=0, mijloc, drepata=n—1;
    while (stinga < =dreapta) {
        mijloc=(stinga+dreapta)/2;
        if( (cond=(*cmp) (key, (BYTE*) baza+mijloc*size))<0)
            dreapta=mijloc—1;
        else
            if (cond>0)
                stinga=mijloc+1;
            else
                return mijloc;
    }
    return —1;
}
```

Singura diferenţă faţă de cazul particular este calculul adresei elementului **baza[mijloc]**: conform definiţiei tablourilor, această adresă este **baza+ +mijloc** $*$ **size**. Deoarece **baza** este de tip **void** $*$, nu i se poate aplica direct operatorul de adunare, drept care este convertit în prealabil la BYTE $*$ (ştim că size este dimensiunea în octeţi).

O **variantă interesantă** este cea care *lucrează intern numai cu pointeri*, deoarece apar probleme la calculul adresei elementului de la mijlocul tabloului. Presupunem în acest caz că funcţia întoarce adresa elementului găsit (de tip **void** $*$) sau NULL.

```
void *bin_2(void *key, void *baza, size_t n, size_t size, PFCMP cmp)
{
    int cond;
    BYTE *stinga=(BYTE*) baza;
    BYTE *mijloc;
    BYTE *dreapta=(BYTE *) baza+(n—1) *size;
    while (stinga<=dreapta)
        mijloc=stinga+(((dreapta—stinga)/size)/2)*size;
        if((cond=(*cmp)(key, mijloc))<0)
            dreapta=mijloc—size;
        else
            if(cond>0)
                stinga = mijloc + size;
            else
                return mijloc;
    }
    return NULL;
}
```

În acest caz, **stinga**, **mijloc** și **dreapta** sunt adresele elementelor corespunzătoare din tablou. Deoarece sunt pointeri de același tip la elementele unui tablou, ei pot fi comparați; are sens deci expresia **(stinga <= dreapta)**. O problemă este calculul lui **mijloc**, deoarece nu putem aduna pointeri. Putem însă să scădem **stinga** din **dreapta**, (indică elementele aceluiași tablou) și obținem distanța în octeți dintre **stinga** și **dreapta** (deoarece sunt de tip BYTE *). Împărțind această valoare la **size**, se obține distanța în indici și apoi, prin împărțire la 2, se obține indicele elementului de la mijloc. Acest indice îl înmulțim acum cu **size** pentru a găsi distanța în octeți față de începutul tabloului. În fine, adunăm această valoare la pointerul **stinga**, obținînd adresa elementului de la mijlocul tabloului.

Chestiunile care trebuie avute deci în vedere la acest tip de probleme sunt corectitudinea operațiilor cu pointeri (comparație, diferență, adunare a unei valori) și calculul corect al adreselor obiectelor procesate.

Pentru un apel comod, se definesc următoarele macroinstrucțiuni (directive **#define** cu parametru):

```
#define NREL (x) (sizeof(x)/sizeof(x[0]))
#define SIZE(x) (sizeof(x[0]))
```

care furnizează numărul de elemente și, respectiv, dimensiunea în octeți a unui element, pentru un tablou x, de tip oarecare.

Iată câteva **exemple de apel**:

a) *Tablou de întregi, cheie întreagă*

```
int a[100];
int cheie=1234;
int găsit_1;
int * găsit_2;
int cmp_num(const void *a, const void *b)
{
    return *((int *) a)—*((int *) b);
}
găsit_1=bin_1(&cheie, a, NREL(a), SIZE(a), cmp_num);
găsit_2=bin_2(&cheie, a, NREL(a), SIZE (a), cmp_num);
```

b) *Tablou de pointeri la* **char**, *cheia este un șir*

```
char *a[100];
char *cheie="1234";
int găsit_1;
char **găsit_2;
int cmp_pstr(const void *key, const void *elem)
{
    return strcmp(key, *((char**) elem));
}
găsit_1=bin_1(cheie, a, NREL(a), SIZE(a), cmp_pstr);
găsit_2=bin_2(cheie, a, NREL(a), SIZE(a), cmp_pstr);
```

Se observă forma modificată a funcției de comparație (în raport cu cea folosită la apelul lui **qsort**). Variabila **key** este un pointer la **char** (adresa obiectului care se caută), iar variabila **elem** este adresa unui element al tabloului (deci de tip **char****). Spre deosebire de funcțiile de comparație de la sortare, unde ambele adrese primite erau de același tip, în cazul căutării cele două variabile pot să difere ca tip (ca în cazul de față).

c) *Tablou de șiruri de lungime constantă, cheia este un șir.*

```
char a[100][80];
char*cheie = "abcdefg";
int găsit_1;
char *găsit_2;
int cmp_str(const void *cheie, const void *elem)
{
    return strcmp (cheie, elem);
}
găsit_1=bin_1(cheie, a, NREL(a), SIZE(a), cmp_str);
găsit_2=bin_2(cheie, a, NREL(a), SIZE(a), cmp_str);
```

În toate cele trei exemple, trebuie observat tipul adecvat al variabilei găsit_2, care este adresa unui element al tabloului în care are loc căutarea.

EXERCIȚII

*32. Scrieți o funcție polimorfică de căutare liniară a unui obiect dat într-un tablou oarecare de elemente de același tip. Prototipul funcției este la fel ca la funcția bin_1. Testați această funcție pe diverse tipuri de tablouri.

*33. Scrieți o funcție polimorfică de căutare liniară, cu același prototip ca în exercițiul 20.32, dar care să interschimbe elementul găsit cu elementul imediat anterior din tablou (în cazul în care obiectul a fost găsit). O asemenea funcție reordonează dinamic tabloul, în sensul că elementele care sunt căutate des și există în tablou, tind să fie aduse pe primele poziții. Astfel, căutările viitoare au șanse să se execute într-un număr mai mic de pași.

CAPITOLUL 21

STRUCTURI

21.1. DEFINIREA ŞI INIŢIALIZAREA STRUCTURILOR. ACCESUL LA MEMBRI

Structura este o *colecţie de una sau mai multe variabile* (numite **membri**), *de diverse tipuri, grupate sub un singur nume.* Operaţiile permise asupra structurilor sunt: atribuirea, accesul la membrii săi (direct şi prin pointeri), luarea adresei, transmiterea la funcţii a unei structuri şi introducerea unei structuri de către o funcţie.

O structură se defineşte prin:

struct nume {declaraţii de variabile};

Iată câteva **exemple:**

```
struct complex {float re; float im;};
struct persoana {char nume[30]; int virsta;};
struct elev {char *nume; int an; char *clasa;};
```

Aceste declaraţii definesc numai tipurile structurilor respective (fără a aloca memorie). Pentru a defini un obiect concret de tipul structurilor respective, cu rezervare de memorie, se poate scrie acum:

```
struct complex z1, z2;
struct persoana x, y, z;
```

În urma acestor declaraţii, z1 şi z2 vor fi 2 structuri de tipul **complex** iar x, y şi z vor fi fiecare structuri de tipul **persoana.** Definirea şi declararea pot fi combinate:

```
struct complex {float re; float im;} z1, z2;
```

situaţie în care se defineşte structura **complex** şi se declară în acelaşi timp z1 şi z2 ca având tipul **struct complex.** Numele structurii ar putea lipsi, dacă definirea cuprinde şi declaraţii:

```
struct { float re; float im; } z1, z2;
```

dar acest lucru nu se recomandă. E bine ca structurile să aibă asociate nume, pentru identificarea tipurilor lor. Este de asemenea recomandabil să se folosească declaraţia **typedef** pentru a lucra mai comod:

```
struct complex { float re; float im; };
typedef struct complex COMPLEX;
```

Putem acum scrie:

```
COMPLEX z1, z2;
```

ceea ce este mult mai sugestiv. O altă formă ar putea fi **combinarea lui typedef** cu definiția structurii:

typedef struct { float re; float im; } COMPLEX;
COMPLEX z1, z2, z3;

Se vede deci că o structură definește un tip de date, putând asocia diverse tipuri de variabile, simple sau structurate, sub un singur tip de date. Întregul grup de variabile poate fi acum tratat global (atribuit, transmis la o funcție etc.), realizându-se astfel o „încapsulare" a datelor.

Structurile se pot inițializa, la declarare, prin precizarea unor valori pentru variabilele care le compun, cuprinse între acolade, de exemplu:

```
COMPLEX z={1.0, —2.5};
struct persoana p={"Popescu Ion", 57};
struct elev e={ "Ionescu Pop", 12, "XIIA"};
```

Dacă structurile conțin elemente mai complexe (tablouri, alte structuri, etc.), se pot folosi mai multe nivele de acolade, de exemplu:

```
struct tip_s {int a[10]; int b[5]; char *s; };
struct tip_s x={
                {1, 2, 3},
                {10, 20, 30, 40, 50},
                "Un şir de caractere"
               };
```

Se inițializează primele 3 elemente ale tabloului **a**, toate cele 5 elemente ale tabloului **b** și șirul **s**.

Toate caracteristicile variabilelor (clasă de alocare, domeniu de vizibilitate etc.) se păstrează și pentru structuri (cu excepția faptului evident că o structură nu poate fi în clasa **register**). Același lucru este valabil și pentru inițializare (structurile în clasa **static** se inițializează doar la încărcarea programului, iar cele din clasa **automatic**, la fiecare intrare în funcția în care sunt declarate).

Accesul direct la membrii unei structuri se face prin operatorul **punct (.)**, aplicat structurii respective:

nume_structura.nume_membru

Iată câteva exemple, în care presupunem definițiile de structuri anterioare:

```
COMPLEX z;
struct persoana x;
struct elev e;
z.re=1.0;
z.im=0.0;
strcpy (persoana.nume, "Ionescu Ion");
persoana.vârsta=20;
e. nume="Popescu Popa";
e. an=10;
e. clasa="XB";
```

În acest context, persoana.nume este un tablou de **char**. Nu putem scrie persoana.nume="..."; (greșit) deoarece nu se pot atribui șiruri de caractere. Putem face însă copiere de șiruri cu **strcpy**. Pe de altă parte, e.nume este un pointer către **char**, deci instrucțiunea:

```
e. nume="Popescu Popa";
```

este o atribuire de pointeri: se atribuie lui **e.nume** adresa șirului constant „Popescu Popa" (care este plasat în memorie la compilare).

E posibil ca o structură să conțină ca membru o altă structură:

```
typedef struct {char *oraş; char *strada; long cod;} ADRESA;
typedef struct {char nume[30]; ADRESA adr;} PERSOANA;
PERSOANA x;
strcpy (x. nume, "Popescu Ion");
x. adr. oraş="Bucureşti";
x. adr.strada="Batiştei 21";
x.adr.cod=73295;
```

Aici **x.adr** este o structură de tipul ADRESA iar **x.adr.oraş** este un membru (de tip pointer la **char**) al structurii **adr**, care este membru al structurii **x**. Declarațiile de mai sus s-ar fi putut scrie, mai confuz, astfel:

```
struct adresa {char *oraş; char *strada; long cod;};
struct persoana {char nume[30]; struct adresa adr;} x;
```

sau, şi mai confuz:

```
struct    persoana {
          char nume[30];
          struct adresa {char *oraş; char *strada; long cod; } adr;
}x;
```

Dacă elementele sunt tipuri structurate (tablouri etc.), accesul se face corespunzător: **x.nume** reprezintă un tablou de 30 de caractere și **x.nume[i]** reprezintă elementul i al tabloului **x.nume**.

Cu structuri se pot construi alte tipuri de date, de exemplu tablouri. Declarația:

```
COMPLEX tab_z[10];
```

definește un tablou de 10 structuri de tip COMPLEX, iar declarația:

```
struct elev tab_e[10];
```

definește un tablou de 10 structuri de tip elev.

Accesul la elemente se face acum ca la tablouri obișnuite. Expresia **tab_z[1].re** va reprezenta câmpul re al celei de-a doua structuri din tabelul **tab_z**, iar expresia: **tab_e[0].nume[2]** va reprezenta al 3-lea caracter din șirul nume al primei structuri din tabloul **tab_e**.

EXERCIȚII

1. Definiți o structură de un anumit tip și afișați dimensiunea ei în octeți, obținută cu operatorul **sizeof**.

2. Folosind tipul COMPLEX, definit mai sus, scrieți un program care să citească partea reală și partea imaginară a unui număr complex, calculând și afișând la consolă modulul și argumentul numărului complex respectiv. Pentru calculul argumentului, se folosește funcția de bibliotecă **atan2**.

3. Definiți un tip de date POLAR, care să reprezinte o structură în care să fie ținute modulul și argumentul unui număr complex. Scrieți și testați secvențe de program care să convertească un număr complex reprezentat prin tipul COMPLEX într-un număr complex reprezentat prin tipul POLAR și reciproc.

4. Definiţi un tablou de 3 elemente de tip COMPLEX, iniţializat corespunzător, şi o secvenţă de program care să calculeze perimetrul triunghiului ale cărui vârfuri sunt cele 3 numere complexe din tablou.

5. Modificaţi structura elev, adăugându-i un element de tip **float**, care să reprezinte o medie şcolară. Să se definească un tablou de structuri de tip elev şi să se citească de la consolă un asemenea tablou, care apoi să se afişeze într-un format corespunzător.

21.2. POINTERI CĂTRE STRUCTURI ŞI ACCESUL PRIN POINTERI. ATRIBUIRI DE STRUCTURI

Un pointer către o structură se declară similar cu pointerii către variabile simple:

COMPLEX z, *pz;

Aici **pz** este un pointer către o structură de tipul COMPLEX. Se poate scrie acum instrucţiunea:

pz=&z;

prin care se dă ca valoare lui pz, adresa structurii z. În operaţiile cu pointeri la structuri (comparaţii, atribuiri etc.), aceştia trebuie să fie către acelaşi tip de structură (acelaşi nume de structură).

În declaraţiile:

```
struct tip_1 {int x; int y;};
struct tip_2 {int x; int y;};
```

tip_1 şi **tip_2** sunt considerate structuri de tipuri diferite, deci

```
struct s1 *p1;
p1= &s2; /*Eroare*/
```

va reprezenta o eroare.

Dacă **p** este un pointer către o structură, atunci ***p** este structura, iar **(*p).nume_membru** este un membru al structurii, ca în exemplul:

```
struct elev e, *pe= &e;
```

prin care se declară o structură **e**, de tip elev şi un pointer **pe** către o structură de tip **elev**, iniţializat cu adresa lui **e**. Se pot face atribuirile:

```
(*pe).nume="Ionescu";
(*pe).an=11;
(*pe).clasa="XIC";
```

Limbajul C permite o formă echivalentă pentru aceste construcţii: dacă **p** este un pointer către o structură, iar **m** este un membru al acestei structuri, atunci **(*p).m** se poate scrie **p->m.** Simbolul -> este format din caracterul - (minus) urmat imediat de caracterul >.

Se pot folosi operatorii ++/-- pentru **p** sau **m**:

(++p)->m : incrementează p şi apoi accesează pe m
(p++)->m : accesează pe m şi apoi incrementează p
(p->m)++ : incrementează pe (p->m) după acces
++(p->m) : incrementează pe (p->m) înainte de acces etc.

Toate chestiunile referitoare la pointeri către variabile simple rămân valabile şi la pointeri către structuri.

Două structuri de același tip pot apare într-o expresie de atribuire, de exemplu:

```
COMPLEX z1, z2, *pz;
z1=z2;
```

Prin definiție o **atribuire de structuri** este o *copiere bit cu bit a elementelor corespunzătoare*. Atribuirea nu are sens decât dacă structurile sunt de același tip. Atribuirea z1=z2 este deci echivalentă cu:

```
z1.re=z2.re;
z1.im=z2.im;
```

Se pot folosi și pointeri, pentru acces indirect;

```
pz=&z2;
z1=*pz;
```

Este de remarcat că această definiție permite unele operații noi. Putem defini o structură cu un singur element, anume un tablou de 80 de caractere:

```
typedef struct {char x[80]; } STRING;
STRING s1, s2;
s1=s2;
```

În urma atribuirii, se vor copia toate cele 80 de elemente ale lui **s2.x** în **s1.x**, ceea ce este echivalent cu:

```
for (i=0; i<80; i++)
    s1.x[i]=s2.x[i];
```

EXERCIȚII

6. Să se scrie un program în care să se definească un pointer la o structură de tip elev, care să fie inițializat cu **malloc**. Pentru calculul dimensiunii se folosește expresia **sizeof** (struct elev). Citiți de la consolă elementele unei asemenea structuri și depuneți-le în structura indicată de pointerul respectiv. (Indicație: spațiul necesar pentru a memora șirurile de caractere către care trebuie să indice membrii structurii se poate rezerva cu strdup).

7. Definiți un tablou de elemente de tip COMPLEX și un pointer la tipul COMPLEX. Parcurgeți și afișați elementele tabloului cu ajutorul pointerului.

8. Definiți două structuri de tipul persoana și scrieți o secvență de program care să copieze element cu element o structură în cealaltă.

9. Rescrieți secvența de la exercițiul 8, folosind atribuirea de structuri.

21.3. STRUCTURI ȘI FUNCȚII

O **structură poate fi transmisă la o funcție** în 3 moduri:

1) *transmițând fiecare element în parte*;
2) *transmițând toată structura (global)*;
3) *transmițând un pointer către structură.*

Cazul 1 înseamnă transmiterea unor variabile simple la funcții și practic, contrazice scopul structurilor: acela de a manipula global o colecție de date. Cazul 1 poate fi folosit explicit, dar nu necesită explicații suplimentare. El poate fi util la construcția unor structuri din elemente simple.

Funcţiile pot întoarce:

I) *o structură (global)*
II) *un pointer la structură*
III) *un singur element al unei structuri*

Când se transmite o structură la o funcţie, transferul se face prin valoare, adică se copiază structura parametru actual în structura locală a funcţiei, bit cu bit.

Similar, la întoarcerea unei structuri de către o funcţie se copiază bit cu bit toate elementele (exact ca la atribuire).

Să considerăm câteva combinaţii posibile :

●a) (1) şi I))

Funcţia care primeşte două numere **float** *şi întoarce o structură* COMPLEX :

```
COMPLEX cons_z (float x, float y)
{
    COMPLEX z;
    z.re=x;
    z.im=y;
    return z;
}
```

Apelul funcţiei este de forma :

```
z=cons_z (1.2, 2.3);
```

●b) (1)+I))

Funcţie de tip struct elev, care primeşte 2 pointeri la **char** *şi un întreg, construind o structură de tip elev*:

```
struct elev cons_e (char *nume, int an, char *clasa)
    {
        struct elev el;
        el. nume=nume; el. an=an;
        el. clasa=clasa;
        return el;
    }
```

Un exemplu de apel poate fi :

```
struct elev e1, e2;
e1=e2=cons_e("Ionescu", 12, "XIID");
```

●c) (2)+II))

Funcţii care primesc o structură de tip COMPLEX, *întorcând elemente simple* **(float)** :

```
float re(COMPLEX z)          float im(COMPLEX z)
{                             {
    return z. re;                 return z.im;
}                             }
```

Exemplele de apel pot fi :

```
z2.re=re(z1);
z2.im=—im(z1);
```

în urma cărora z2 va fi conjugatul lui z1.

d) (2)+1))

Funcţia care primeşte o structură COMPLEX *şi întoarce o structură* COMPLEX:

```
COMPLEX conj(COMPLEX z)
{
    COMPLEX w;
    w.re=z.re;
    w.im=—z.im;
    return w;
}
```

cu apeluri de forma:

```
z2=conj(z1);
z1=conj (z1);
```

Este de remarcat faptul că, *prin structuri, putem realiza transmiterea prin valoare a unor obiecte care în mod normal sunt transmise prin referinţă.* Cazul tipic este cel al tablourilor cu 1 dimensiune (la care totdeauna se transmite adresa de început).

De exemplu, funcţia:

```
void conv (char a[ ])
{
    int i;
    for (i=0; a[i]!='\0'; i++)
        a[i]=toupper (a[i]);
}
```

converteşte şirul primit la litere mari. Dacă se consideră secvenţa:

```
char *s="abcd";
conv (s);
```

după apel, s va conţine „ABCD". Să includem acum, şirul într-o structură, definind:

```
typedef struct {char alfa [30]; } STRING;
STRING s={"abcd"};
```

şi să rescriem funcţia **conv** astfel:

```
void conv (STRING a)
{
    int i;
    for (i=0; a.alfa [i]!='\0'; i++)
        a. alfa[i]=toupper (a. alfa[i]);
}
```

După un apel de tipul:

```
conv (s);
```

s este nemodificat (va conţine tot „abcd"), deoarece s-au convertit caracterele din copia locală (din funcţia **conv**) a structurii s, fără efect asupra lui s (practic, s-a realizat transmiterea prin valoare a şirului s către funcţia **conv**). Putem scrie însă funcţia **conv**, cu structuri, astfel:

```
STRING conv (STRING a)
{
    int i;
    STRING b;
    for (i=0; a.alfa[i]!='\0'; i++)
        b. alfa[i]=toupper (a.alfa[i]);
    return b;
}
```

cu un apel de forma :

```
s=conv (s);
```

în urma căruia au loc două copieri ale întregului tablou : o dată, la apel, se copiază întregul tablou **s.alfa** în variabilă locală **a.alfa**, şi o dată la atribuire, când se copiază întregul tablou **b.alfa**, acum modificat, înapoi în **s.alfa**. De fapt, au loc 3 atribuiri, deoarece instrucţiunea **return b** este tradusă într-o copiere a structurii **b** într-o zonă locală (în stivă) şi apoi urmează copierea în structura **s**.

Când structurile sunt mari (ca în exemplul anterior), este ineficient să se transmită prin valoare (copiere) întreaga structură. Este *mult mai eficient să se transmită un pointer către acea structură, iar în interiorul funcţiei să se folosească accesul prin pointer.*

•e) (3)

Funcţia care realizează „conjugarea" unei structuri COMPLEX, *primind un pointer la acea structură :*

```
void bara (COMPLEX *p)
{
    p->im=-p->im;
}
```

având apelul de forma:

```
bara (&z);
```

•f) (3)

Funcţia care preia de la tastatura elementele unei structuri de tip struct elev, primind un pointer la structura respectivă şi actualizând corespunzător structura :

```
void get_elev_1 (struct elev *p)
{
    char buf[80];
    printf ("Nume elev:");
    p->nume=strdup (gets (buf));
    printf ("Anul:");
    scanf("%d", &(p->an));
    printf („Clasa:");
    p->clasa=strdup (gets (buf));
}
```

cu apel de forma :

```
struct elev el;
get_elev_1 (&el);
```

În structura **elev**, **nume** şi **clasa** sunt pointeri la **char**. Şirurile introduse trebuie memorate undeva şi adresele de memorare se pun în aceşti pointeri.

Varianta fără pointeri este :

```
struct elev get_elev_2 (void)
{
    struct elev e;
    char buf [80];
    printf ("Nume elev:");
    e. nume=strdup (gets(buf));
    printf("Anul:");
    scanf("%d", &(e.an));
    printf("Clasa:");
    e.clasa=strdup(gets (buf));
    return e;
}
```

cu apelul corespunzător :

```
struct elev el;
el=get_elev_2 ();
```

Această variantă e ineficientă: se copiază întreaga structură, deci un consum mare de timp. În prima funcție, era definit doar un pointer și nu se făcea nici o copiere.

Pointerii la structuri trebuie obligatoriu folosiți atunci când se dorește ca o funcție să modifice parametrii ei formali, deci pentru realizarea transmiterii prin referință a unei structuri la funcție.

Este util și în cazul în care funcția întoarce un pointer către structură, folosit în special la crearea dinamică de obiecte de tip structură (crearea lor în faza de execuție, nu prin declararea la compilare).

De exemplu, avem declarația unui pointer pe la structură, dar nu avem spațiu rezervat pentru structură, ci numai pentru pointerul pe. Putem aloca spațiu pentru o structură cu **malloc** :

```
struct elev *pe=(struct elev*) malloc (sizeof (struct elev));
if (pe==NULL)
    /*eroare*/
else {
    /*putem folosi acum structura*/
    pe—>nume=...
    pe—>an=...
}
```

Este indicat să se folosească întotdeauna sizeof și nu să calculăm noi care e dimensiunea unei structuri. De exemplu, structura :

struct a { int x ; char c ; } ;

(în ipoteza sizeof(int) = =2), ar putea ocupa 4 octeți, nu 3, pe anumite mașini unde există restricții de aliniere (de exemplu, întregii pe 2 octeți să se afle neapărat la adrese pare). Aceasta se datorează faptului că, dacă se declară un tablou de asemenea structuri (care, prin definiție, înseamnă elemente aflate la adrese succesive de memorie), restricția poate fi îndeplinită numai dacă structura ocupă un număr par de octeți.

EXERCIȚII

10. Să se scrie o funcție care, primind o structură de tip elev, afișează la consolă, într-un format corespunzător, elementele structurii.

11. Folosind tipul COMPLEX, să se definească o serie de funcții pentru operațiile uzuale cu numere complexe : modul, argument, parte reală, parte imaginară, sumă, diferență, produs, cât, ridicat la putere, conversie din forma algebrică în forma trigonometrică și reciproc.

12. Să se definească o structură adecvată pentru a memora vectori în spațiul cartezian cu trei dimensiuni. Folosind structura respectivă, să se scrie funcții de calcul pentru produsul unui vector cu un scalar, al produsului scalar și al produsului vectorial al doi vectori.

13. Să se definească o structură adecvată pentru a memora cercuri în spațiul cu două dimensiuni (de exemplu, se pot memora coordonatele centrului și raza). Să se scrie o funcție care să testeze apartenența unui punct dat la un cerc dat, întorcând puterea punctului față de cerc.

21.4. STRUCTURI CA ELEMENTE ALE UNOR ALTE TIPURI DE DATE. STRUCTURI CU AUTOREFERIRE

Cu structuri sau cu pointeri la structuri se pot forma acum alte tipuri de date: tablouri de structuri, tablouri de pointeri la structuri, tablouri multidimensionale cu elemente structuri etc.

Iată câteva **exemple**:

```
typedef struct point {int x; int y;} POINT;
void move (POINT);
void draw (POINT);
POINT p[ ]={
            {10, 10},
            {10, 20},
            {30, 40}
};
void poligon (POINT p[ ], int n)
{
      int i;
      move (p[0]);
      for (i=0; i<n; i++)
            if (i==n—1)
                  draw (p[0]);
            else
                  draw (p[i+1]);
}
```

Presupunem că funcțiile **move** și **draw** poziționează spotul pe ecran, respectiv trasează un segment din punctul curent în punctul specificat. Funcția **poligon** primește un tablou de n POINT-uri și va uni cele n puncte cu segmente de dreaptă.

Un caz important îl reprezintă **structurile cu autoreferire**, adică acele *structuri care au ca membru unul sau mai mulți pointeri către același tip de structură.*

O structură nu se poate conține pe ea însăși ca element, declarația:

```
struct a { int x; struct a y; };      /* eroare */
```

fiind greșită, dar ea poate conține un pointer către același tip de structură:

```
struct a { int x; struct a *next; };      /* corect */
```

sau către alt tip de structură, chiar dacă al doilea tip nu este încă definit:

```
struct a { struct b *pb; };
struct b { struct a *pa; };
```

Structurile cu autoreferire (care conțin un pointer către același tip de structură) *permit implementarea tipurilor de date înlănțuite:* stive, cozi, liste, liste dublu înlănțuite, arbori, grafuri etc.

O **listă liniară** este o succesiune de elemente aflate la adrese oarecare de memorie, în care fiecare element conține, pe lângă alte informații, adresa de memorie a următorului element. În limbajul C, listele liniare *se implementează prin structuri cu autoreferire, unul din membrii structurii fiind un pointer la același tip de structură.* Adresa elementului următor al listei se mai numește și **câmp de legătură**, iar restul elementelor din structură alcătuiesc **câmpul de informație**. Ultimul element al listei conține valoarea NULL în câmpul de legătură. O listă este transmisă la o funcție printr-un pointer la primul element. O listă vidă va fi semnalată printr-un pointer NULL.

Crearea unei liste liniare dintr-un şir de caractere se poate face atât în varianta recursivă (**str_to_list_r**), cât şi iterativă (**str_to_list_i**). Ambele funcţii întorc un pointer la primul element al listei sau NULL, dacă lista generată este vidă. Funcţia **new_el** generează spaţiu pentru un element, iniţializându-l cu argumentele primite şi întoarce un pointer la spaţiul generat.

Funcţia **print_list** tipăreşte o listă:

```
#include <stdio.h>
#include <stdlib. h>
struct  el{char d; struct el *next;};
typedef struct el ELEMENT, *LINK;
LINK new_el (char c, LINK p)
{
    LINKt =(LINK) malloc (sizeof (ELEMENT));
    if (t==NULL) {
        puts ("\nEroare malloc...\n");
        exit (1);
    }
    t—>d=c;
    t—>next=p;
    return t;
}
```

• **Varianta iterativă** de generare a unei liste dintr-un şir de caractere foloseşte două variabile pointer de tip LINK, t şi p. Prima, iniţializată cu NULL, se foloseşte pentru a memora adresa primului element, iar a doua pentru a parcurge elementele pe măsură ce acestea se generează:

```
LINK str_to_list_i (char *s)
{
    LINKt =NULL, p;
    if (*s!=0)
        p=t=new_el(*s++, NULL);
    else
        return NULL;
    while (*s) {
        p—>next=new_el(*s++, NULL);
        p=p—>next;
    }
    return t;
}
```

• **Varianta recursivă** testează întâi cazul de bază (şirul primit este vid), întorcând pointerul NULL. Dacă şirul nu este vid, se apelează **new_el** cu parametrii *s (primul caracter din şir) şi **str_to_list_r**(s+1), adică un pointer la o listă creată de aceeaşi funcţie, dar cu restul de caractere din şir. Funcţia **new_el** va crea un element pe baza acestor două argumente, întorcând un pointer la acest element. Acest pointer este întors programului apelant.

```
LINK str_to_list_r(char *s)
{
    if (*s=='\0')
        return NULL;
    else
        return new_el (*s, str_to_list_r (s+1));
}
```

```
void print_list (LINK t)
{
    if (t==NULL)
        printf ("NULL\n");
    else {
        putchar (t->d);
        printf("->");
        print_list (t->next);
    }
}
```

● **Formele generale** ale unei funcţii de prelucrare a listelor liniare, în varianta iterativă şi recursivă sunt următoarele (se presupune o funcţie **prel_elem** care prelucrează un element al listei):

```
void prel_ite (LINK t)
{
    while (t!=NULL) {
        prel_elem(t);
        t=t->next;
    }
    prel_elem (NULL);
}
void prel_rec(LINK t)
{
    prel_elem (t);
    if(t!=NULL)
        prel_rec (t->next);
}
```

EXERCIŢII

14. Să se scrie o funcţie recursivă care să tipărească elementele unei liste liniare, în ordinea inversă decât funcţia **print_list**. (*Indicaţie:* se modifică locul unde apare apelul recursiv în funcţie).

15. Să se scrie o funcţie (atât în varianta recursivă, cât şi iterativă) care să întoarcă numărul de elemente ale unei liste liniare. Funcţia trebuie să întoarcă valoarea 0 în cazul listei vide.

***16.** Să se scrie o funcţie care, primind o listă dată (printr-un pointer la primul element), realizează o duplicare a listei, creând o altă listă, cu acelaşi număr de elemente, fiecare element având acelaşi conţinut în câmpul de informaţie cu elementul corespunzător din lista dată. Funcţia trebuie să întoarcă un pointer la primul element al listei nou create.

*** 17.** Să se scrie o funcţie care, primind doi pointeri de tip LINK (către două liste date), realizează concatenarea celei de-a doua liste la prima listă. Concatenarea se face modificând câmpul de legătură al ultimului element din prima listă. Funcţia trebuie să întoarcă un pointer la lista concatenată rezultată. Să se verifice cazurile de excepţie (unul sau ambii pointeri sunt NULL-i).

21.5. TABLOURI DE STRUCTURI

Să definim acum o structură de forma:

typedef struct { int key; char *şir; } STRUC;

şi un tablou de structuri:

STRUC tab[100];

Vrem să căutăm un element în tablou, după cheia numerică **key**, folosind funcţiile polimorfice **bin_1** sau **bin_2**, definite în capitolul anterior. Definim funcţia de **comparaţie:**

```
int cmp_struc_1 (const void *key, const void *elem)
{
    return *((int *) key)—((STRUC*) elem)—>key;
}
```

Aici **key** este adresa unui întreg (cheia numerică), iar **elem** este adresa unui element al tabloului, deci adresa unei structuri de tipul STRUC, deci e vorba de tipuri diferite. Se converteşte pointerul **key** la **(int *)** şi apoi se ia conţinutul, apoi se converteşte pointerul **elem** la **(STRUC *)** şi se ia câmpul **key** al structurii indicate de **elem**. Se întoarce diferenţa acestor doi întregi.

Exemplu de apel :

```
STRUC   a[100];
int     key=5;
int     found_1;
STRUC   *found_2;
found_1=bin_1( &key, a, 100, sizeof (STRUC), cmp_struc_1);
found_2=bin_2( &key, a, 100, sizeof (STRUC), cmp_struc_1);
```

Dorim acum să sortăm tabloul de structuri, folosind funcţia de bibliotecă **qsort**, după cheia numerică **key**. Trebuie modificată corespunzător funcţia de comparaţie, deoarece se primesc acum adresele a două obiecte de acelaşi tip (două elemente ale tabloului):

```
int cmp_struc_2(const void *a, const void *b)
{
    return ((STRUC *)a)—>key — ((STRUC *)b)—>key;
}
```

Dacă se doreşte ca, la chei numerice egale, să se sorteze după şirul din structură (în ordine alfabetică), atunci funcţia de comparaţie trebuie scrisă de forma:

```
int cmp_struc_3(const void *a, const void *b)
{
    STRUC *p=(STRUC *) a;
    STRUC *q=(STRUC *) b;
    int dif;
    return (dif=p—>key—q—>key) ? dif : strcmp(p—>şir, q—>şir);
}
```

Apeluri posibile ale lui **qsort** sunt în acest caz:

```
qsort (a, 100, sizeof (STRUC), cmp_struc_2);
qsort (a, 100, sizeof (STRUC), cmp_struc_3);
```

Mai **trebuie reţinut** că *două structuri nu se pot compara, dar se pot scrie funcţii de comparaţie,* ca în exemplele precedente.

EXERCIȚII

18. Definiți un tablou de structuri de tip elev, inițializat într-un mod corespunzător și sortați acest tablou în diverse moduri: în ordine alfabetică a numelui, în ordine crescătoare a anului, etc. Definiți o funcție de comparație adecvată pentru fiecare situație.

19. Definiți un tablou de structuri de tip persoana și sortați tabloul în ordinea alfabetică a numelor. Căutați apoi o anumită persoană în tablou, după nume, folosind funcția de căutare binară.

21.6. UNIUNI. CÂMPURI DE BIȚI

Uniunile sunt *structuri care pot conține (la momente de timp diferite), obiecte de tipuri diferite*. Practic, este vorba de mai multe variabile, suprapuse în același spațiu de memorie. Putem privi problema și invers, ca având o zonă de memorie pe care o interpretăm ca un anumit tip de variabilă la un moment dat și ca un alt tip, la alt moment. Folosirea uniunilor asigură gestiunea corectă a spațiului de memorie (dimensiune, aliniere etc.).

Declarația unei uniuni este similară cu cea a structurilor, **de exemplu**:

```
union alfa {int, ival; float fval; char *sval; };
union alfa u;
```

sau, folosind declarația typedef:

```
typedef union (int ival; float fval; char *sval; } ALFA;
ALFA u;
```

Accesul la membrii uniunii este similar cu cel de la membrii unei structuri:

```
u. ival     /*se interpretează u ca o variabilă int*/
u. fval     /* . . . . . . . . . . . . . . . . . . .float*/
u. sval     /* . . . . . . . . . . . . . . . . . . . .char* */
```

Nu este indicat să facem presupuneri despre cum sunt interpretate fiecare dintre aceste variabile în raport cu spațiul de memorie. Dacă vrem dimensiunea uniunii, atunci putem folosi **sizeof(union alfa)** sau **sizeof(ALFA)**. Dimensiunea nu este totdeauna maximul dimensiunilor membrilor uniunii, așa cum la structuri nu era neapărat suma dimensiunilor membrilor structurii.

La fel ca la structuri, se pot defini *pointeri la uniuni*, **de exemplu:**

```
ALFA *pu;
pu—>ival        /*referiri la */
pu—>fval        /*elementele uniunii */
pu—>sval        /*indicate de pu*/
```

Toate operațiile permise la structuri sunt permise și la uniuni: atribuiri, transfer ca parametri la funcții, întoarceri de către funcții, pointeri la uniuni etc.

Pe scurt, o **uniune** este o *structură în care toți membrii au deplasament 0 față de adresa de început a structurii, structura e suficient de mare ca să conțină cel mai mare membru și sunt respectate restricțiile de aliniere pentru toți membrii.*

Când se scrie ceva într-un membru al uniunii, ceea ce se va citi apoi trebuie să fie de același tip, altfel rezultatele depind de implementare. Programatorul trebuie să mențină gestiunea a ce conține uniunea în mod curent

(de exemplu un **întreg**, un **float**, un **char ***). De obicei se ține o variabilă suplimentară care spune ce conține curent uniunea. *Structurile pot apare în uniuni și reciproc. Oricare combinație poate apare în tablouri:*

```
struct {
        char *name; int flag; int utype;
        union {int ival; float fval; char *sval; } u;
} tab[10];
```

Aici tab este tablou de structuri, care au un element uniune. **Exemple de** *acces* pot fi:

tab[3].nume	(char *)
tab[2].flag	(int)
tab[5].u	(uniune)
tab[5].u.ival	(câmpul ival din uniunea u)
*tab[i].u.sval	(primul caracter al lui sval din structura i)
tab[i].u.sval[0]	(la fel)

Câmpul **utype** poate fi folosit pentru a memora ce conține curent uniunea.

Inițializarea unei uniuni se poate face numai cu o valoare de tipul primului membru al ei.

Iată un **exemplu** *de folosire a uniunilor:* o funcție care tipărește reprezentarea internă a tipului float.

```
void bin_float (float x)
{
        union {float f; unsigned char c; } u;
        size_t n=sizeof (float);
        unsigned char *p= &(u. c);
        u.f=x;
        while (n— —)
                printf ("%02x", *p++);
}
```

Se suprapune practic octetul c peste primul octet din f și se accesează la nivel de octet. Funcția va lucra corect pentru orice implementare a tipului **float**. Schimbând peste tot **float** cu **double**, se va obține reprezentarea binară pentru **double**.

Câmpurile de biți apar în structuri în care se specifică numărul de biți pe care este reprezentată fiecare variabilă (care poate fi numai de tip integral, deci **char**, **int**, etc.). Câmpurile de biți sunt utile la economie de spațiu sau pentru interfața cu dispozitive de intrare/ieșire (în general pentru acces la nivel de bit).

Iată un **exemplu**:

```
struc ss {
        unsigned a: 1;
        unsigned b: 3;
        unsigned  : 4;
        unsigned c: 3;
        unsigned d: 2;
} s;
```

va corespunde la Borland C unei reprezentări de forma celei din figura 21.1.

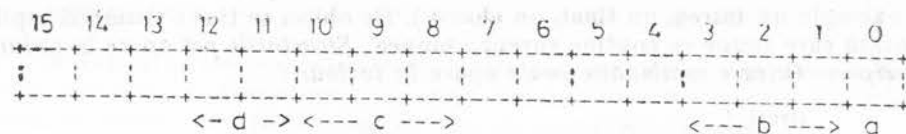


Fig. 21.1.

Din dimensiunile variabilelor, rezultă:

```
s.a:poate lua va'orile 0...1
s.b:poate lua valorile 0...7
s.c:poate lua valorile 0...7
s.d:poate lua valorile 0...3
```

Dacă se dorește stocarea comodă a unui întreg (pe 16 biți) în această structură, se poate defini o uniune de forma:

```
union uu {
    struc ss s;
    unsigned short w;
} u:
```

Se poate atribui acum un întreg acestei uniuni și apoi se folosesc câmpurile de biți, de exemplu pentru tipărire:

```
unsigned short w;
u.w=w;
printf ("d=%d c=%d b=%d a%d\n", u.s.d, u.s.c, u.s.b, u.s.a);
```

Această tehnică ne scutește de a face filtrări incomode ale biților respectivi prin operații logice și de deplasare.

Reprezentarea internă a structurilor cu câmpuri de biți este dependentă de implementare (alocarea variabilelor de la stânga la dreapta sau invers, dimensiunea maximă permisă a structurii, dimensiunea maximă a unui câmp etc.).

Aceste tipuri speciale de structuri (împachetate) simplifică lucrul la nivel de bit, dar în general codul generat este destul de voluminos (ceea ce se câștigă în volumul de date se pierde în volumul de cod generat).

EXERCIȚII

20. Să se scrie o funcție care să tipărească (în hexazecimal) reprezentarea internă a variabilelor de tip **double**.

***21.** Microprocesorul 8086 are un registru de flaguri (indicatori) care se poziționează corespunzător în urma instrucțiunilor mașină aritmetice și logice sau care controlează anumite operații ale procesorului. Configurația registrului de flaguri este dată în figura 21.2.

Flagurile sunt denumite astfel:

- **OF (Overflow Flag)**: este 1 dacă operația a condus la un transport/împrumut înspre/dinspre b.c.m.s. al rezultatului dar nu la un transport/împrumut din/în b.c.m.s. al rezultatului.

- **DF (Direction Flag)**: precizează sensul operațiilor cu șiruri de caractere (adresele cresc sau scad).

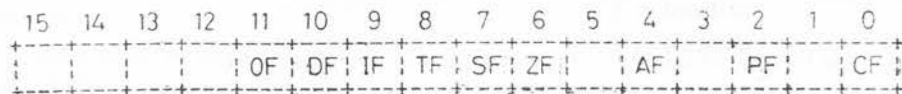


Fig. 21.2.

• **IF (Interrupt Flag)** : precizează starea bistabilului de întreruperi al procesorului; valoarea 1 permite întreruperi externe.

• **TF (Trace Flag)**: poziţionat la 1, permite execuţia pas cu pas (la nivel de instrucţiuni maşină) a unui program.

• **SF (Sign Flag)** : este 1 dacă bitul cel mai semnificativ al rezultatului unei operaţii aritmetice este 1.

• **ZF (Zero Flag)** : este 1 dacă rezultatul unei operaţii aritmetice/logice este nul.

• **AF (Auxilliary Carry)** : este 1 dacă în operaţia care s-a efectuat a fost un transport înspre bitul 4 sau un împrumut dinspre bitul 4 al rezultatului.

• **PF (Parity Flag)** : este 1 dacă suma modulo 2 a celor mai puţin semnificativ 8 biţi ai rezultatului este 0.

• **CF (Carry Flag)** : este 1 dacă operaţia a condus la un transport/împrumut din/în bitul cel mai semnificativ al rezultatului.

Definiţi o structură cu câmpuri de biţi pentru a gestiona aceste flaguri. Următoarea funcţie (care foloseşte unele instrucţiuni maşină şi variabila _AX, predefinită la Borland C) întoarce conţinutul registrului de flaguri al procesorului, filtrând biţii care nu au semnificaţie (adică forţându-i la 0):

```
typedef unsigned short WORD ;
WORD flags (void)
{
    asm {pushf; pushf; pop ax; }
    _AX &=0x0fd5;
    asm popf;
    return _AX;
}
```

Folosind funcţia de mai sus, să se scrie o funcţie cu tipul **void,** fără parametri, care să tipărească explicit starea tuturor flagurilor, din momentul apelului funcţiei, în forma:

OF = , DF = , IF = ,etc.

Pentru a nu altera flagurile procesorului în interiorul acestei funcţii, se folosesc instrucţiunile **asm pushf** ca primă instrucţiune în funcţie şi, respectiv, **asm popf** ca ultimă instrucţiune. De asemenea, nu se folosesc variabile locale în funcţie, ci numai variabile externe.

Pentru a putea stoca un **WORD** în structura cu câmpuri de biţi, se poate defini o uniune cu doi membri: structura cu câmpuri de biţi şi un **WORD.**

CAPITOLUL 22

INTRĂRI/IEȘIRI ȘI FUNCȚII DE BIBLIOTECĂ

22.1. GENERALITĂȚI

Bibliotecile standard C oferă o mare varietate de funcții pentru operații de intrare/ieșire, orientate pe fișiere, ca și o clasă largă de funcții generale. Dispozitivele periferice ale sistemului sunt văzute tot ca fișiere, cu identificatori predefiniți.

Funcțiile de I/E se pot clasifica după diverse criterii. Astfel, există *funcții de citire/scriere la nivel de caracter* (octet), *la nivel de șir de caractere* (linie de text), funcții de I/E *cu conversie de format*, funcții *la nivel de înregistrări* (fără conversie de format), funcții *pentru acces direct*, etc.

Limbajul C și-a propus să asigure un set de funcții care să fie ușor de implementat pe orice sistem, respectând aceeași sintaxă.

Independența totală de sistemul de operare nu se realizează totdeauna (la MS-DOS există, de exemplu, anumite detalii de implementare care nu pot fi ocolite, cum ar fi modul text/modul binar), dar o bună parte din funcții este universală (există în orice implementare). Există două clase mari de funcții de I/E numite **funcții standard** și **funcții de sistem.** Funcțiile de sistem corespund implementării C sub sistemul de operare UNIX dar ele coincid practic cu funcțiile DOS orientate pe handlere (versiuni DOS>=3.0).

Prototipurile funcțiilor de bibliotecă sunt cuprinse în **fișiere header** cu nume predefinite. În continuare, vor fi prezentate principalele funcții ale bibliotecii standard, specificându-se și fișierul header unde sunt declarate prototipurile.

22.2. INTRĂRI/IEȘIRI STANDARD : (stdio.h)

22.2.1. Accesul la fișiere

Conceptul de bază de la intrări/ieșiri standard este cel de **pointer de fișier.** În orice implementare este definit (cu **typedef**) un tip de structură, numit **FILE**. Nu interesează ce conține concret structura, deoarece funcțiile folosesc pointeri către tipul **FILE.** Fișierele disc și dispozitivele de intrare/ieșire standard sunt gestionate în mod univoc prin variabile de tip **FILE ***, care se asociază în mod unic la fiecare fișier prelucrat, pe durata prelucrării. Dispozitivele de intrare/ieșire standard au asociate permanent câte un ase-

menea pointer, cu nume predefinit. Definirea concretă a tipului **FILE** este în fișierul header **stdio. h**. Declararea unui pointer la fișier se face deci prin
FILE *fp;

Se prezintă în continuare funcțiile de intrare/ieșire standard.

a. *Deschiderea fișierelor*

● **FILE *fopen(const char *filename, const char *mode);**

Unica operație prin care se poate atribui o valoare corectă unui pointer la fișier este operația de **deschidere**, care se face prin apelul funcției **fopen**. Prin deschidere, *se stabilește o conexiune logică între variabila pointer și un anumit fișier*. Din acest moment, toate prelucrările fișierului se vor face prin pointerul la fișier obținut la deschidere.

Parametrul **filename** este o specificare pentru **numele fișierului**, iar **mode** este un șir de caractere care descrie **modul de acces**. Apelul funcției se face prin:

fp=fopen(......);

Dacă operația a decurs corect, pointerul întors de **fopen** este diferit de **NULL**. Dacă, din diverse motive, deschiderea nu s-a putut face corect, **fopen** întoarce **NULL**, fapt ce trebuie testat prin program înainte de a trece la alte prelucrări.

Numele fișierului depinde ca formă de sistemul de operare. De exemplu, la MS-DOS, numele poate conține o specificare de cale **(path)**. Trebuie acordată atenție caracterului '\', care trebuie dublat într-un șir constant de caractere. De exemplu, pentru a specifica fișierul DOS cu numele:

c:\user\file.dat

șirul de caractere trebuie scris în forma:

"c:\\user\\file.dat"

Modurile posibile de acces sunt:

— **"r"**: deschidere (în mod text) pentru citire;

— **"w"**: deschidere (în mod text) pentru scriere. Dacă fișierul exista anterior, conținutul vechi se pierde. Dacă nu există un fișier cu numele specificat, se creează un fișier nou;

— **"a"**: deschidere (în mod text) pentru adăugare. Dacă fișierul nu exista anterior, se creează un fișier nou. Dacă există, se va scrie la sfârșitul său;

— **"r+"**: deschidere (în mod text) pentru actualizare (punere la zi) adică citire și scriere;

— **"w+"**: deschidere (în mod text) pentru actualizare (punere la zi). Dacă fișierul exista anterior, conținutul vechi se pierde. Dacă nu există un fișier cu numele specificat, se creează un fișier nou;

— **"a+"**: deschidere (în mod text) pentru actualizare, cu scriere la sfârșitul fișierului.

Modurile care conțin + (actualizare) permit scrierea și citirea din același fișier, mai precis prin același pointer la fișier.

Dacă se pune sufixul **b**, de exemplu **"rb"** sau **"w+b"**, se indică **modul binar**. Se poate pune explicit și sufixul **t**, pentru a indica explicit **modul text**. Distincția dintre binar și text depinde de sistemul de operare (la MS-DOS există diferențe). Modul text este implicit. Unele implementări (de exemplu Borland) permit schimbarea modului de acces implicit, prin intermediul unei variabile externe (**fmode** la Borland C), de aceea este indicat să prevedem explicit modul text sau binar.

Constanta predefinită **FILENAME_MAX** precizează lungimea maximă a numelui unui fişier, iar constanta **FOPEN_MAX** precizează numărul maxim de fişiere care pot fi deschise simultan.

Sunt deschise permanent fişierele cu pointeri predefiniţi următori:

— **stdin** (uzual asignat la consolă intrare);

— **stdout** (uzual asignat la consolă ieşire);

— **stderr** (totdeauna asignat la consolă ieşire).

În diverse implementări există şi alte fişiere standard predefinite, de exemplu, la Borland C:

— **stdaux** (uzual asignat la interfaţa serială COM1);

— **stdprn** (uzual asignat la imprimanta LPT1).

Fişierele **stdin** şi **stdout** pot fi redirectate către alte dispozitive sau fişiere de exemplu din linia de comandă. O lansare în execuţie cu comanda:

>nume_prg > file_out.dat

va face ca toate datele care se afişau din programul **nume_prg** în mod normal la consolă să fie acum scrise în fişierul **file_out.dat.** Similar, o comandă de forma:

> nume_prg < file_in.dat

va înlocui toate citirile de la consolă din programul **nume_prg** cu citiri din fişierul **.file_in.dat.**

Parametrii de redirectare (cei precedaţi de < sau de >) nu sunt număraţi ca argumente ale programului în linia de comandă (în **argc**) şi nu apar în **argv** şi **env**. Am presupus **main** de forma:

void main (int argc, char **argv, char **env);

Redirectarea fişierelor standard este o proprietate a sistemului de operare şi nu are legătură cu limbajul C. La MS-DOS se poate face redirectare la orice program executabil, care foloseşte funcţii DOS de intrare/ieşire orientate pe handlere de fişier.

b. *Reasignarea fişierelor (dispozitivelor) prin program*

● **FILE *freopen(const char *filename, const char *mode, FILE *fp);**

În urma apelului, se închide fişierul **fp**, se deschide fişierul cu numele **filename** în modul de acces **mode**, atribuind pointerul la acest fişier lui **fp.** Se întoarce această valoare atribuită lui **fp**, care poate fi şi **NULL**, dacă deschiderea nu s-a putut face corect. Faptul că se întoarce aceeaşi valoare permite testarea corectitudinii operaţiei. Această funcţie permite deci redirectarea dispozitivelor prin program.

Iată un exemplu de program sub MS_DOS care va redirecta **stdout** la un fişier disk şi apoi din nou la consolă (dispozitivul consolă este denumit în MSDOS CON):

```
#include <stdio.h>
void main (void)
{
        printf ("Acest text se va tipări la consolă\n");
        if (freopen ("file_out.dat", "wt", stdout)==NULL){
            fprintf (stderr, "eroare la reasignarea lui stdout\n") ;
            exit (1);
        }
        printf ("Acest text se va tipări în fişierul file_out.dat\n") ;
        fclose (stdout);
```

```
    if (freopen ("CON", "wt", stdout)==NULL){
        fprintf (stderr, "eroare la reasignare lui stdout la CON\n");
        exit(1);
    }
    printf ("Acest text se va tipări din nou la consolă\n");
}
```

Se redirectează **stdout** la fişierul disk „**file_out.dat**", care este deschis în modul „**wt**", testându-se dacă valoarea întoarsă de **freopen** nu este **NULL**. Din acest moment, **stdout** este asociat cu fişierul disk specificat şi toate scrierile care se făceau în mod normal la consolă, se vor face acum în acest fişier. Se închide acest fişier şi apoi se redeschide **stdout**, asociindu-l cu fişierul (dispozitivul) MS-DOS „CON", (consola). După această operaţie, afişările se vor face normal.

c. *Forţarea scrierii bufferelor în fişierele deschise pentru scriere*

● **int fflush(FILE *fp);**

Dacă **fp** este un pointer la un fişier deschis pentru scriere sau punere la zi, se forţează scrierea datelor nescrise din bufferul asociat în fişier. Întoarce **EOF** în caz de eroare sau 0 în cazul normal.

În standardul ANSI, efectul este nedefinit dacă fişierul nu a fost deschis pentru scriere sau punere la zi. Unele implementări permit apelul acestei funcţii şi pentru fişiere deschise pentru citire, efectul fiind ştergerea buffer-ului asociat fişierului.

d. *Închiderea unui fişier*

● **int fclose(FILE *fp);**

Funcţia întoarce **EOF** în caz de eroare sau 0 în caz normal.

Prin închidere, încetează conexiunea logică între pointer şi fişier, care fusese stabilită la deschidere. Concret, *efectele închiderii* sunt:

— scrierea datelor nescrise din buffer-ul de ieşire în fişier (la fişiere deschise pentru scriere/actualizare);

— abandonarea datelor necitite din buffer-ul de intrare (la fişiere deschise pentru citire/actualizare);

— eliberarea buffer-elor alocate automat;

— tăierea conexiunii logice între pointer şi fişier.

e. *Ştergerea unui fişier*

● **int remove(const char *filename);**

Funcţia întoarce o valoare diferită de 0 în caz de eroare sau 0 în caz normal, efectul fiind ştergerea fişierului cu numele specificat.

f. *Schimbarea numelui unui fişier*

int rename(const char *new_name, const char *old_name);

Efectul este evident. Se întoarce ceva diferit de 0 în caz de eroare.

g. *Crearea unui fişier temporar*

FILE *tmpfile(void);

Se creează un fişier temporar în modul "**wb+**", care va fi şters automat la închiderea sau la terminarea normală a programului. Funcţia întoarce un pointer la acel fişier sau **NULL** în caz de eroare. Apeluri succesive vor deschide fişiere distincte.

22.2.2. **Distincţia dintre modurile text şi binar la MS-DOS**

Diferenţele se referă la două aspecte şi anume: tratarea caracterelor '\n' şi tratarea sfârşitului de fişier.

a. *Tratarea caracterului '\n'*

Caracterul '\n' are codul ASCII 0xA, deci este caracterul **linefeed**.

● **În modul text**, lucrurile se petrec astfel:

— la scriere în fişier, '\n' se converteşte într-o secvenţă de 2 caractere CR şi LF (0xD şi 9xA);

— la citire din fişier, o secvenţă CR, LF este convertită într-un singur caracter: '\n' (LF). (fig. 22.1).

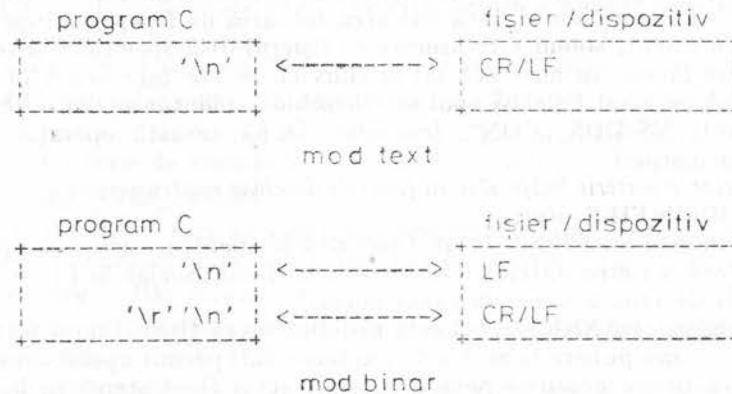


Fig. 22.1.

Aceste conversii au loc şi la operaţiile cu periferice (consolă etc.).

● **În modul binar**, aceste conversii nu au loc, deci o secvenţă CR, LF va fi citită ca 2 caractere distincte.

Aceste abordări diferite inseamnă, de exemplu, că un program care citeşte şi numără caracterele dintr-un fişier text poate conduce la rezultate diferite, după cum modul de lucru este binar sau text.

Trebuie menţionat că, practic, toate driverele de citire de la consolă, din diverse sisteme de operare, generează, la apăsarea tastei ⟨CR⟩ (Enter), o secvenţă de caractere CR/LF, care va fi apoi convertită în '\n' (consola este în mod text). Implementările uzuale oferă şi funcţii de citire caracter la nivel fizic (de exemplu **getch()** la Borland C), care ocolesc driverele sistemului de operare şi buffer-ele acestuia. Cu o asemenea funcţie, la apăsarea tastei ⟨Enter⟩, se va obţine caracterul '\r' (CR).

Scrierea datelor într-un fişier poate fi făcută în format intern sau extern (cu conversie de format). O variabilă **int**, care este memorată uzual pe 2 octeţi poate fi scrisă în format intern (se scriu efectiv cei doi octeţi) sau în format extern (se scrie reprezentarea ASCII, deci cifre zecimale sau în alta bază specificată). De exemplu, întregul 2573 va fi scris în format intern ca 2 octeţi cu codurile 0xD şi 0xA, adică tocmai codurile ASCII pentru CR şi LF. Evident că, dacă vom scrie în modul binar, fără conversie de format şi vom citi în modul text, aceşti 2 octeţi vor fi convertiţi la unul singur (0xA) şi rezultatele vor fi eronate.

Ca **regulă generală utilă**, este indicat ca *scrierea şi citirea să se realizeze cu acelaşi tip de funcţii:* **fprintf/fscanf** sau **fputs/fgets** sau **fwrite/fread** etc. Dacă fişierul din care se citeşte nu este creat prin program, trebuie alese funcţii de citire adecvate, depinzând de conţinutul fişierului.

b. *Tratarea sfârşitului de fişier*

În ambele moduri (text şi binar), funcţiile de I/E gestionează numărul de octeţi din fişier la citire şi semnalează corect sfârşitul de fişier. De exemplu, funcţia **fgetc()**, la terminarea fişierului, întoarce o valoare întreagă predefinită, **EOF**, care este de obicei —1. Acest întreg **EOF** nu există de fapt în fişier, ci este generat de **fgetc()**. În modul text însă, există un caracter special, **0x1A**, ca ultim caracter al fişierului (există fizic în fişier). La întâlnirea acestui caracter, **fgetc()** va întoarce **EOF** (fig. 22.2). Caracterul **0x1A** poate fi generat de la tastatura apăsând CONTROL şi Z; din acest motiv, el este uneori notat cu CTRL/Z sau ^Z. Această abordare este moştenită de la sistemul CP/M, specific maşinilor de 8 biţi. Este evident că, dacă avem un fişier care conţine numere întregi în reprezentare internă sau numere reale, poate apare un octet cu valoarea 26 (**0x1A**). Dacă se citeşte în modul text, se constată că nu se poate trece de acest octet, el fiind interpretat ca sfârşit fizic de fişier. Citirea trebuie făcută în acest caz în modul binar.

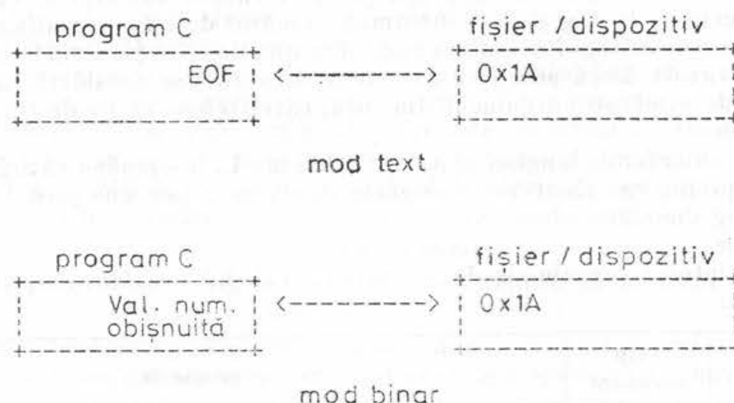


Fig. 22.2.

22.2.3. Funcţii de intrare/ieşire cu conversie de format

a. *Scriere cu format (fprintf, printf, sprintf, vfprintf, vprintf, vsprintf)*

• **int fprintf(FILE *fp, const char *format,...);**

Punctele (...) arată că este vorba de o *funcţie cu număr variabil de parametri.* Argumentele specificate se convertesc şi se scriu în **fp**, conform cu şirul **format**. Funcţia întoarce numărul de caractere scrise sau o valoare negativă în caz de eroare. Formatul conţine caractere obişnuite, care sunt scrise în fişier ca atare şi descriptori de format (de conversie), fiecare dintre aceştia din urmă cauzând conversia şi scrierea unui argument. Numărul de descriptori trebuie să coincidă cu numărul de argumente.

Fiecare descriptor începe cu caracterul % şi se termină cu caracterul de conversie. Forma generală este:

• **%[flags][width][.prec][lnmod]type**

cerinţele cuprinse între [] fiind opţionale.

• **flags**

— : aliniere la stânga a argumentului în cadrul câmpului

\+ : numerele vor fi tipărite obligatoriu cu semn

0 : completare la stânga cu zerouri (la numere)

blank : dacă primul caracter nu e semnul, se va adăuga un ' '

#: forma alternată pentru scriere: pentru %o se scrie un 0 inițial, pentru %x sau %X se scrie 0x sau 0X dacă valoarea nu este 0, pentru %e, %E, %f, %g, %G se scrie punctul zecimal obligatoriu, iar pentru %g și %G nu se elimină zerourile de la sfârșit.

- **width**

este un număr care specifică lățimea minimă a câmpului. Argumentul convertit va fi scris pe un câmp de această lățime sau mai mare, dacă este necesar. Dacă sunt mai puține caractere, se va completa câmpul cu blancuri la stânga (implicit) sau la dreapta (dacă s-a specificat flagul —). Dacă s-a specificat flagul 0, se completează cu 0 la stânga. Dacă **width** este caracterul *, atunci se consideră că lățimea este dată de următorul argument din listă, care trebuie să fie de tip **int**.

- **.prec**

este un număr care specifică precizia; la %s înseamnă numărul maxim de caractere ce se va tipări; la %e, %E și %f înseamnă numărul de cifre după punctul zecimal; la %g și %G înseamnă numărul de cifre semnificative, iar la descriptori de întregi înseamnă numărul minim de cifre (se scriu 0-uri în fața dacă este cazul). Dacă **prec** este caracterul *, atunci se consideră că lățimea este dată de următorul argument din listă, care trebuie să fie de tip **int**.

- **lnmod**

este un specificator de lungime și poate fi : h, l sau L, însemnând că argumentul este interpretat ca: short sau unsigned short, long sau unsigned long, respectiv long double.

- **type**

este descriptorul propriu-zis de conversie. Tabelul următor cuprinde toți descriptorii

Caracter	Tip argument	Se convertește la
d, i	**int**	Notație zecimală cu semn
0	**int**	Notație hexa fără semn (fără 0 inițial).
x, X	**int**	Notație hexa fără semn (fără 0x (0X) inițial). se folosesc abcdef pentru x sau ABCDEF pentru X
u	**int**	Notație zecimală fără semn
c	**int**	Un caracter (după conversia argumentului la unsigned char).
s	**char** *	Se tipăresc caractere din șir, până la '\0' sau un număr de caractere indicat de precizie.
f	**double**	Notație zecimală de forma : [—]mmmm.ddd unde d este dat de precizie (implicit 6). Precizia 0 suprimă punctul zecimal.
e, E	**double**	[—]m.ddde+/—xx sau [—]m.dddE+/—xx, unde numărul de d-uri este dat de precizie (similar cu f).
g, G	**double**	Similar cu %e sau %E dacă exponentul este < —4 sau >=precizia; altfel, similar cu %f. 0-urile și punctul zecimal de la sfârșit nu se scriu.
p	**void** *	Se tipărește argumentul ca pointer
n	**int***	Numărul de caractere scrise până în momentul respectiv se înscrie în argumentul specificat.
%		Se tipărește '%'. Nu se face nici o conversie.

● **int printf(const char *format,...);**
este echivalent cu: fprintf(stdout, format,...);

● **int sprintf(char *s, const char *format,...);**
este similar cu **printf**, cu excepţia faptului că se scrie în şirul de caractere s, adăugându-se '\0' la sfârşit. Şirul trebuie dimensionat suficient. Caracterul '\0' nu se socoteşte la numărul de caractere scrise care este întors de către funcţie.

● **int vfprintf(FILE *fp, const char *format, va_list arg);**
este similar cu **fprintf**, dar foloseşte mecanismul specific funcţiilor cu număr variabil de argumente (vezi 22.7).

● **int vprintf(const char *format, va_list arg);**
este similar cu **printf**, dar foloseşte mecanismul specific funcţiilor cu număr variabil de argumente.

● **int vsprintf(char *s, const char *format, va_list arg);**
este similar cu **sprintf**, dar foloseşte mecanismul specific funcţiilor cu număr variabil de argumente.

b. *Citire cu format* **(fscanf, scanf, sscanf)**

● **int fscanf(FILE *fp, const char *format, ...);**
Se citesc date din fişierul **fp**, sub controlul şirului **format** şi se atribuie valorile convertite argumentelor, care *trebuie să fie obligatoriu pointeri*. Citirea se încheie când se epuizează şirul de formate. Se întoarce **EOF** dacă apare sfârşit de fişier înainte de a se face vreo conversie şi atribuire, sau dacă apare o eroare. Altfel, se întoarce numărul de elemente convertite corect şi atribuite (deci **nu** numărul de octeţi). Formatul poate conţine:

— blank-uri sau tab-uri, care sunt ignorate;

— caracter obişnuit (fără %), care este aşteptat să coincidă cu primul caracter diferit de spaţii albe din fişier;

— specificatori de conversie, constând din %, caracterul opţional * (care suprimă asignarea), un număr opţional care specifică lăţimea maximă a câmpului şi un caracter opţional (**h**, **l** sau **L**) care specifică dimensiunea variabilei (la fel ca la **printf**).

Rezultatul conversiei este plasat în variabila indicată de pointerul respectiv din lista de argumente. Dacă se indică suprimarea asignării cu *, câmpul următor de intrare este sărit. Un câmp de intrare este definit ca un şir de caractere diferite de spaţii albe. El se consideră fie până la următorul spaţiu alb, fie până la epuizarea lăţimii câmpului (dacă este specificată). Aceasta înseamnă că se va citi şi peste delimitatorii de linie ('\n'). Spaţiile albe sunt: ' ', '\t', '\n', '\r', '\v' şi '\f'. Caracterele de conversie **d**, **i**, **n**, **o**, **u** şi **x** pot fi precedate de **h**, dacă argumentul este un pointer de tip **short** * în loc de **int** *, sau de **l**, dacă argumentul este pointer de tip **long** *. Caracterele de conversie **e**, **f**, şi **g** pot fi precedate de **l**, dacă pointerul este de tip **double** *, sau de **L**, dacă este de tip **long double** *.

Tabelul următor descrie toţi specificatorii de conversie.

Caracter	Tip argument	Date de intrare
d	**int***	Întreg zecimal,
i	**int***	Întreg octal (cu 0) sau hexa (cu 0x sau 0X) sau zecimal
o	**int***	Întreg octal, cu sau fără 0 iniţial.

Caracter	Tip argument	Date de intrare
u	**unsigned int** *	Întreg zecimal fără semn.
x	**int** *	Întreg hexa, cu sau fără 0x sau 0X inițial.
c	**char** *	Caractere. Următoarele caractere sunt plasate în șirul specificat, în număr câte sunt date de lățime (implicit 1). Nu se adaugă '\0'. Nu se sare peste spații albe. Pentru a citi următorul caracter diferit de spațiu alb, se folosește %1s.
s	**char** *	Șir de caractere diferite de spațiu alb. Se adaugă '\0'.
e, f, g	**float** *	Numere în virgulă mobilă. Formatul cuprinde semn opțional cifre cu un punct zecimal (opțional), exponent opțional e/E urmat de un întreg, cu semn opțional.
p	**void** *	Valoarea unui pointer așa cum e tipărită de: **printf** ("%p", ...).
n	**int** *	Numărul de caractere citite până în momentul respectiv se înscrie în argumentul specificat. Nu se modifică numărul de elemente citite și atribuite.
[...]	**char** *	Se citește cel mai lung șir de caractere care apar în mulțimea specificată între [ ]. Se adaugă '\0'. Forma [ ]...] include [în mulțimea caracterelor.
[^...]	**char** *	Similar cu [...]. Se citesc caracterele care nu apar în mulțimea specificată între [ ]. Se adaugă '\0'. Forma [^] ..] include] în mulțimea caracterelor.
%		Caracterul %. Nu se face nici o atribuire.

- **int scanf(const char *format,...);**

este echivalent cu: **fscanf(stdin, format,...);**

- **int sscanf(char *s, const char * format,...);**

este similar cu **fscanf**, cu excepția faptului că se citește din șirul de caractere **s**.

Trebuie remarcat că la funcția **printf**, care are numărul și tipul argumentelor neprecizate (în afară de primul), au loc conversii implicite de tip ale argumentelor care sunt în număr variabil. Astfel, are loc promovarea la tipul **int** pentru toate tipurile integrale, iar toate argumentele de tip **float** sunt convertite la **double**. Promovare la tipul **int** înseamnă concret: un caracter, un întreg scurt sau un câmp întreg de biți, cu sau fără semn, ca și un obiect de tip enumerare, sunt convertite la tipul **int**, dacă acesta poate reprezenta valoarea tipului original, sau la tipul **unsigned int**, în caz contrar.

Aceste conversii explică de ce, în tabelul de specificatori de format pentru **printf**, %f corespunde tipului **double**. Regulile de conversie implicită de mai sus se aplică la toate funcțiile cu număr variabil de argumente.

22.2.4. Funcții de intrare/ieșire la nivel de caracter

● **int fgetc(FILE *fp);**
Întoarce următorul caracter din **fp**, ca un **unsigned char**, convertit la **int**, sau **EOF**, dacă s-a întâlnit sfârșit de fișier sau în caz de eroare.

● **char *fgets(char *s, int n, FILE *fp);**
Se citesc cel mult **n**−**1** caractere din **fp** în tabloul **s**, citirea oprindu-se la '\n'. Caracterul '\n' este inclus în **s** și apoi se adaugă '\0'. Se întoarce **s** sau **NULL** în caz de sfârșit de fișier sau eroare.

● **int fputc(int c, FILE *fp);**
Se scrie **c** (convertit la **unsigned char**) în **fp**. Se întoarce **c** sau **EOF** în caz de eroare.

● **int fputs(const char *s, FILE *fp);**
Se scrie **s** în fișier (nu e obligatoriu să conțină '\n'). Întoarce **EOF** în caz de eroare sau ceva >=0 în rest.

● **int getc(FILE *fp);**
Este echivalent cu **fgetc**, cu excepția faptului că este macroinstrucțiune și nu funcție. E posibil ca **fp** să fie evaluat de mai multe ori.

● **int getchar(void);**
Este echivalent cu **getc(stdin)**.

● **char *gets(char *s);**
Citește următoarea linie de la consolă și o pune în **s**. Caracterul '\n' nu se include în **s**. Se adaugă '\0'. Întoarce **s** sau **NULL** dacă a fost sfârșit de fișier (CTRL/Z) sau în caz de eroare.

● **int putc(int c, FILE *fp);**
Similar cu **fputc**, dar este macroinstrucțiune.

● **int putchar(int c);**
Este echivalent cu **putc(c, stdout)**.

● **int puts(const char *s);**
Scrie **s** la consolă. Adaugă '\n'. Întoarce **EOF** dacă a fost eroare, altfel ceva >=0.

● **int ungetc(int c, FILE *fp);**
Se pune **c** (convertit la **unsigned char**) în bufferul asociat fișierului **fp**, astfel încât următorul caracter citit din **fp** va fi **c**. Se poate pune înapoi numai un caracter înainte de a face alte citiri. **EOF** nu se poate pune la loc. Funcția întoarce **c** sau **EOF** în caz de eroare.

22.2.5. Funcții de intrare/ieșire orientate pe înregistrări

Aceste funcții citesc/scriu din/în fișiere fără nici o conversie și fără a se face vreo interpretare a datelor. La scriere, în fișier va apare reprezentarea internă a datelor (imaginea memoriei). Se folosesc cu modul de acces binar. Noțiunea de bază este cea de *înregistrare* (zona compactă de octeți) care se citește/scrie. Citirea/scrierea se face în poziția curentă din fișier care poate fi modificată cu funcții speciale. După citire/scriere, poziția în fișier este actualizată automat, încât să se indice următoarea înregistrare. Înregistrările pot conține date de orice fel.

● **size_t fread(void *ptr, size_t size, size_t nrec, FILE *fp);**
Se citesc cel mult **nrec** înregistrări din fișierul **fp**, presupuse de lungime **size** (în octeți) în tabloul **ptr**. Se întoarce numărul de înregistrări citite, care poate fi mai mic strict decât **nrec**, sau chiar 0, caz în care s-a atins sfârșitul de fișier (uzual) sau este eroare. Se pot folosi funcțiile **feof** și **ferror** pentru a face distincție între aceste cazuri.

● **size_t fwrite(const void *ptr, size_t size, size_t nrec, FILE *fp);**

Scrie **nrec** înregistrări de lungime **size**, de la adresa **ptr**, în fișierul **fp**. Se întoarce numărul de înregistări scrise care este mai mic decât **nrec** numai în caz de eroare.

22.2.6. Funcții de control al poziției în fișier și de eroare

La fișierele exploatate în acces direct, putem controla poziția în fișier, deci poziția din care se va citi, respectiv în care se va scrie. Poziționarea se face la nivel de octet.

● **int fseek(FILE *fp, long offset, int origin);**

Se setează poziția în fișierul **fp**. În cazul unui fișier binar, poziția va fi setată la **offset** octeți față de **origin**, care poate fi una din constantele predefinite **SEEK_SET** (față de începutul fișierului), **SEEK_END** (față de sfârșitul fișierului) sau **SEEK_CUR** (față de poziția curentă). La fișiere text, **offset** trebuie să fie 0 sau o valoare întoarsă de **ftell()** (caz în care **origin** trebuie să fie **SEEK_SET**). Se întoarce 0 în caz de eroare.

● **long ftell(FILE *fp);**

Se întoarce poziția curentă în fișierul **fp** sau **—1L** în caz de eroare.

● **void rewind(FILE *fp);**

Este echivalent cu: **fseek(fp, 0L, SEEK_SET); clearrer(fp);** și înseamnă poziționare la începutul fișierului (rebobinare).

● **int fgetpos(FILE *fp, fpos_t *ptr);**

Memorează poziția curentă din fișier în ***ptr**, care va putea fi folosit ulterior de către **fsetpos**. Tipul predefinit **fpos_t** este adecvat acestor valori. Se întoarce 0 în caz de eroare.

● **int fsetpos(FILE *fp, const fpos_t *ptr);**

Se setează poziția curentă din fișierul **fp** la valoarea ***ptr**, memorată anterior cu **fgetpos**. Se întoarce 0 în caz de eroare.

● **void clearerr(FILE *fp);**

Se șterg indicatorii de sfârșit de fișier și de eroare asociați fișierului **fp**.

● **int feof(FILE *fp);**

Se întoarce ceva **!=0** dacă este poziționat indicatorul de sfârșit de fișier asociat lui **fp**.

● **int ferror(FILE *fp);**

Se întoarce ceva **!=0** dacă este poziționat indicatorul de eroare asociat lui **fp**.

● **void perror(const char *s);**

Tipărește la **stderr** șirul **s** și un mesaj ce depinde de implementare, funcție de ultima eroare intîlnită. Variabila întreagă externă **errno**, declarată în ⟨**errno.h**⟩ poate conține un cod care descrie ultima eroare de execuție.

A se vedea și funcția **strerror** din 22.4.

22.2.7. Programe de intrare/ieșire

Funcțiile de bibliotecă standard oferă posibilitatea dezvoltării comode de programe portabile, care să realizeze operații de intrare/ieșire.

O primă **aplicație** este un *program de copiere a unui fișier în alt fișier*, care va fi dezvoltat în diverse variante. Presupunem numele programului **cp** și considerăm că el primește argumentele în linia de comandă.

O primă variantă este:

```
#include <stdio.h>
void main (int argc, char *argv[ ])
{
    FILE *sursă, *dest;
    int c;
    if (argc!=3) {
        fprintf (stderr, "cp: Sintaxa este : cp sursa dest\n");
        exit (1);
    }
    if ((sursa=fopen (argv [1], "rt"))==NULL) {
        fprintf (stderr, "cp: Eroare deschidere fișier %s\n", argv[1]) ;
        exit (1);
    }
    if ((dest=fopen (argv [2], "wt"))==NULL) {
        fprintf (stderr, "cp: Eroare deschidere fișier %s\n", argv [2]) ;
        exit (1);
    }
    c=fgetc (sursa);
    while (c!=EOF) {
        fputc(c, dest);
        c=fgetc (sursa);
    }
    fclose (sursa);
    fclose (dest);
}
```

Varianta de mai sus *realizează copierea la nivel de octet*. Se verifică dacă numărul de argumente este corect (deci să fie prezente atât numele fișierului sursă, cât și al celui destinație) și apoi se deschid cele două fișiere în modul text, cu verificarea corectitudinii operațiilor. Se intră apoi într-o buclă de citire-scriere, care se execută cât timp nu s-a ajuns la sfârșitul fișierului sursă, fapt semnalat prin întoarcerea valorii **EOF** de către fgetc.

Această variantă de program este adecvată fișierelor text (care conțin caractere ASCII) și va eșua (în general) la MS-DOS, în cazul fișierelor oarecare, datorită interpretării incorecte a unui posibil octet cu valoarea 0x1a ca sfârșit de fișier.

• *O altă variantă* de program de copiere fișiere text *poate folosi funcțiile de citire/scriere la nivel de linie de text*. Trebuie rezervat un buffer de dimensiune acoperitoare pentru citirea unei linii. Bucla de citire-scriere se realizează cu **fgets/fputs** și continuă cât timp funcția **fgets** întoarce ceva diferit de NULL. Și această variantă va funcționa corect numai cu fișiere text.

```
#include <stdio. h>
char buffer [1000];
void main (int argc, char *argv[ ])
{
    FILE *sursa, *dest;
    char *s;
    if (argc !=3) {
        fprintf (stderr, "cp: Sintaxa este : cp sursă dest\n");
        exit (1);
    }
```

```
if ((sursa=fopen (argv[1], "rt"))==NULL) {
        fprintf (stderr, "cp: Eroare deschidere fişier %s\n", argv [1])
        exit (1);
    }
    if ((dest=fopen (argv[2], "wt"))==NULL) {
        fprintf (stderr, "cp: Eroare deschidere fişier %s\n", argv [2]);
        exit (1);
    {
    s=fgets (buffer, 1000, sursă);
    while (s!=NULL) {
        fputs (buffer, dest);
        s=fgets (buffer, 1000, sursă);
    }
    fclose (sursă);
    fclose (dest);
}
```

• *A treia variantă de program* foloseşte funcţiile de citire/scriere orientate pe înregistrări. Fişierele se deschid în modul binar, iar înregistrările se consideră de lungime 1 octet. Această variantă funcţionează corect cu orice tip de fişier.

```
#include <stdio.h>
char buffer [1000];
void main (int argc, char *argv [ ])
{
    FILE *sursă, *dest;
    size_t n;
    if (argc!=3) {
        fprintf (stderr, "cp: Sintaxa este : cp sursă dest\n");
        exit (1);
    }
    if ((sursă=fopen (argv[1], "rb"))==NULL) {
        fprintf (stderr, "cp: Eroare deschidere fişier %s\n", argv[1]);
        exit(1);
    }
    if ((dest=fopen (argv [2], "wb"))==NULL) {
        fprintf (stderr, "cp: Eroare deschidere fişier %s\n", argv [2]);
        exit(1);
    }
    n=fread (buffer, 1, 1000, sursă)·
    while (n>0) {
        fwrite (buffer, 1, n, dest);
        n=fread (buffer, 1, 1000, sursă);
    }
    fclose (sursă);
    fclose (dest);
}
```

Operaţiile de intrare/ieşire cu conversie de format se realizează cu funcţiile **fprintf** şi **fscanf**, care operează exact la fel ca **printf** şi **scanf** cu deosebirea că scriu/citesc în/din fişiere disc în loc de a scrie/citi la/de la consolă.

• Funcţia următoare **write_tab** scrie un *tablou a de întregi, de dimensiune n, într-un fişier specificat, cu conversie de format.*

```
void write_tab (int a[ ], int n, char *fişier)
{
    int i;
    FILE *fp=fopen (fişier, "wt");
    if (fp==NULL) {
        fprintf (stderr, "write_tab: eroare deschidere fişier %s\n",
            fişier);
        exit (1);
    }
    for (i=0; i<n; i++)
        fprintf (fp, "%6d%c", a[i]),
            ((i%10)==0 || i==n—1) ?'\n' : ' ');
    fclose (fp);
}
```

● *Funcţia* următoare **read_tab** *citeşte cel mult n întregi dintr-un fişier specificat într-un tablou a, întorcând numărul de întregi efectiv citiţi.*

```
int read_tab (int a[ ], int n, char *fişier)
{
    int i=0, m, x;
    FILE*fp=fopen (fişier, „rt");
    if (fp==NULL) {
        fprintf (stderr, "read_tab: eroare deschidere fişier %s\n",
            fişier);
        exit (1);
    }
    while (i<n) {
        m=fscanf (fp, "%d", &x);
        if (m>0)
            a[i++]=x;
        else
            break;
    }
    fclose (fp);
    return i;
}
```

Deoarece nu se cunoaşte aprioric numărul de întregi din fişier, trebuie prevăzută testarea explicită a sfârşitului de fişier. Se foloseşte faptul că fscanf întoarce numărul de obiecte corect citite, convertite şi asignate. Dacă acest număr este pozitiv, deci s-a citit corect, se atribuie lui a[i] valoarea citită în variabila intermediară x, incrementând totodată pe i. În caz contrar se iese din bucla **while**, iar valoarea lui i (care reprezintă numărul de elemente ale tabloului care au fost asignate) este întoarsă programului apelant. Condiţia i<n de la while asigură citirea a cel mult n întregi.

● *Următoarea funcţie de citire foloseşte funcţiile de poziţionare în fişiere binare orientate pe înregistrări, citind înregistrarea cu numărul i.* Se presupune că dimensiunea unei înregistrări este size. Funcţia întoarce —1 dacă nu s-a putut face corect poziţionarea şi citirea din fişier şi 0 în rest. Înregistrarea citită se depune la adresa dest.

```
int read_bin (void *dest, size_t size, size_t i, char *file)
{
    size_t err;
    FILE *fp=fopen (file, "rb");
```

```
    char *buf=(char *) malloc (size);
    if (fp==NULL || buf==NULL)
        return —1;
    if (fseek(fp, i*size, SEEK_SET)) {
        fclose (fp);
        free (buf);
        return —1;
    }
    err=fread (buf, size, 1, fp);
    if (err==1)
        memcpy (dest, buf, size);
    free (buf);
    fclose (fp);
    return (err) ? 0:—1;
}
```

Prelucrările de fişiere din exemplele precedente erau fie de tip citire, fie de tip scriere. Uneori este necesar ca în acelaşi fişier deschis să se facă atât operaţii de citire, cât şi de scriere. Un asemenea mod de prelucrare se numeşte **actualizare** sau **punere la zi.** Pentru aceasta, la deschiderea fişierului cu **fopen()**, se va specifica un mod conţinând caracterul +.

● Să considerăm un *program care acceptă un şir de caractere şi nume generice de fişiere în linia de comandă şi realizează o codificare a acestora,* în sensul că fiecărui octet din fişier i se aplică o funcţie de codificare. Funcţia de codificare se bazează pe o cheie întreagă (pe un octet), obţinută din şirul de caractere introdus. Pentru uşurinţă, alegem funcţia de codificare bijectivă şi coincizând cu inversa sa. Astfel, dacă se mai execută o dată programul cu un fişier codificat, se realizează de fapt decodificarea acestuia.

Se vede că succesiunea de operaţii trebuie să fie : citire date din fişier, codificare, scriere date codificate în fişier. Specific este aici faptul că citirea şi scrierea trebuie să se facă din/în acelaşi fişier. Programul sursă este listat în continuare.

```
#include <stdio. h>
#include <dir.h>
#include <string. h>
#define NR_BYTES 512
typedef unsigned char BYTE;
BYTE        mask;
static      BYTE buf[NR_BYTES];
BYTE crypt(BYTE c)
{
    return c ^ mask;
{
void prel_file (char *name)
{
    FILE *fp;
    int m, n, i;
    fpos_t pos1, pos2;
    if ((fp=fopen (name, "rb+"))==NULL) {
        fprintf (stderr,
            "\ncrypt: Eroare deschidere fişier %s\n", name);
        return;
    }
```

```
do {
     fgetpos(fp, &pos1);
     n=fread (buf, 1, NR_BYTES, fp);
     fgetpos (fp, &pos2);
     fsetpos (fp, &pos1);
     if (n==0)
          break;
     for (i=0; i<n; i++)
          buf [i]=crypt (buf[i]);
     m=fwrite (buf, 1, n, fp);
     fsetpos (fp, &pos2);
     if (m!=n) {
          fprintf (stderr, "\ncrypt: Eroare scriere in %s\n", name);
          fclose (fp);
          return;
     }
} while (n==NR_BYTES);
     fclose (fp);
     printf ("\n\t %s", name);

void main (int argc, char *argv [ ])
{
     int          done;
     struct       ffblk work;
     BYTE         buf [20];
     int          i;
     if (argc!=3)  {
          printf ("\ncrypt: Sintaxa este: crypt <key> <file (s)>\n");
          exit (1);
     }
     strncpy (buf, argv [1], 19);
     buf [19]='\0';
     for (i=0, mask=0; buf [i]!='\0'; i++)
          mask+=buf [i];
     if (mask==0) {
          printf ("\ncrypt: Cheia %s va lăsa fişierele neschimbate",
               argv [1]);
          printf ("\n Încercaţi altă cheie\n");
          exit (0);
     }
     done=findfirst (argv [2], &work, 0);
     if (done==—1) {
          printf ("\ncrypt%s: Nu există fişiere\n", argv [2]);
          exit (0);
     }
     do {
          if (done==—1)
               break;
          prel_file (work. ff_name);
          done=findnext ( &work);
```

```
    } while (!done);
    putchar ('\n'):
}
```

Se definește la nivel exterior bufferul **buf** de dimensiune **NR_BUF** și un octet **mask**, care se folosește la codificare.

Funcția **crypt** primește un octet și îl codifică la nivel de bit, folosind **mask**. Se verifică ușor că aplicarea de două ori succesiv a acestei funcții unui octet îl lasă neschimbat.

Funcția **prel_file** codifică un fișier, primind numele său **name** ca parametru. Se deschide fișierul respectiv în modul **rb+**, adică actualizare în mod binar (fișierul trebuie să existe deja). Modurile de tipul **w+** creează un fișier pentru actualizare (dacă fișierul există, conținutul său este pierdut). Dacă este eroare la deschidere, se afișează un mesaj și se revine în programul apelant.

În continuare se intră într-o buclă de tip citire-prelucrare-scriere. Problema care apare este că la funcțiile **fread** și **fwrite**, în urma unei operații de citire/scriere, se deplasează un indicator logic în fișier, astfel încât următoarea cirire/scriere să se facă „în continuare". Poziția curentă în fișier se poate însă citi și modifica cu ajutorul funcțiilor **fgetpos** și **fsetpos**. Tipul predefinit de date **fpos_t** este adecvat pentru valorile vehiculate de aceste două funcții.

Bucla începe cu memorarea poziției curente în fișier în variabila **pos1**, apoi cu citirea a maxim **NR_BUF** octeți și memorarea noii poziții (după citire) în variabila **pos2**. Funcția **fread** întoarce numărul de înregistrări efectiv citite (în cazul de față, numărul **n** de octeți). Se poziționează acum indicatorul logic al fișierului la poziția **pos1**, adică revenim la poziția de unde s-au citit cei **n** octeți. Dacă **n** este 0, s-a ajuns la sfârșitul fișierului și se iese din buclă. Dacă nu, se codifică toți cei **n** octeți din **buf** și se scriu în fișier, poziționând apoi indicatorul la **pos2**, deci după octeții citiți. Dacă **m** (numărul de octeți scriși efectiv) diferă de **n**, înseamnă că a fost o eroare la scriere și se revine în programul principal, după afișarea unui mesaj adecvat și închiderea fișierului. Bucla continuă cât timp **n** este egal cu NR_BYTES, adică nu s-a ajuns la sfârșitul fișierului. La sfârșitul prelucrării, se afișează la consolă numele fișierului.

Programul principal folosește două funcții nestandard pentru acces la directorul curent. Funcția **find_first** (care se apelează o singură dată) primește un nume de fișier care poate conține caracterele * sau ? și poziționează câmpul **ff_name** din structura predefinită **ffblk** cu numele complet al unui fișier care corespunde numelui generic primit. Dacă nu există asemenea fișiere, funcția întoarce —1. Funcția **find_next** primește adresa unei structuri de tip **ffblk** poziționată anterior cu **find_first**, întorcând în câmpul **ff_name** numele complet al următorului fișier care corespunde numelui generic. Când nu mai sunt asemenea fișiere, **find_next** întoarce —1.

Calculul lui **mask** se face prin sumarea modulo 256 a caracterelor din șirul introdus pe post de cheie. Dacă valoarea rezultă 0, caz în care funcția **crypt** va lăsa neschimbat octetul pe care îl codifică, se afișează un mesaj adecvat și se iese din program. Altfel, se prelucrează toate fișierele specificate.

• **Exemple de folosire a programului:**

> crypt aaaa file.dat ⟨CR⟩
(codifică fișierul file.dat după cheia „aaaa")
> crypt ababab *.c ⟨CR⟩

(codifică toate fişierele cu extensia .c, după cheia „ababab")
> crypt 123456 a*.* ⟨CR⟩
(codifică toate fişierele care încep cu litera a, după cheia „123456")
> crypt 123456 a*.* ⟨CR⟩
(decodifică fişierele codificate prin comanda anterioară)

EXERCIŢII

1. Completaţi cele 3 variante de program de copiere fişiere, cu testarea eventualelor erori care pot apare la operaţiile de scriere în fişier.

2. Scrieţi un program de test pentru funcţia **read_tab**, prevăzând toate situaţiile posibile (fişierul conţine un număr mai mic, mai mare de întregi decât cel cerut pentru citire, fişierul este vid etc.).

3. Rescrieţi funcţiile **read_tab** şi **write_tab** cu ajutorul funcţiilor de bibliotecă **fread**/**fwrite**, deci fără conversie de format. Dimensiunea unei înregistrări va fi **sizeof(int)**.

4. Definiţi o structură oarecare şi funcţii adecvate pentru scriere/citire în/din fişiere a unui tablou de structuri, folosind funcţiile **fread/fwrite**. Se va considera că înregistrarea din fişier este chiar imaginea de memorie a structurii respective.

*5. Scrieţi un program de afişare la consolă, care să permită opţiuni de afişare în ASCII, în hexazecimal sau în octal. Opţiunile se dau în linia de comandă prin secvenţe de forma —a, —x sau— o. Prevedeţi o opţiune implicită. Numele fişierului se introduce tot în linia de comandă.

22.3. TESTAREA APARTENENŢEI LA CLASE DE CARACTERE : ⟨ctype. h⟩

Aceste funcţii primesc ca parametru un întreg care poate fi EOF sau un întreg convertibil la **unsigned char** şi întorc un întreg diferit de 0 dacă este satisfăcută condiţia de apartenenţă, respectiv 0, în caz contrar.

iscntrl(c) : 1 dacă c este caracter de control (0x0..0x1f şi 0x7f)
isdigit(c) : 1 dacă c este cifră zecimală ('0'..'9')
isxdigit(c) : 1 dacă c este cifră hexa ('0'..'9' sau 'a'..'f' sau 'A'..'F')
isgraph(c) : 1 dacă c este afişabil, fără spaţiu (0x21..0x7e)
islower(c) : 1 dacă c este literă mică (0x61..0x73) ('a'..'z')
isupper(c) : 1 dacă c este literă mare (0x41..0x53) ('A'..'Z')
isprint(c) : 1 dacă c este afişabil, cu spaţiu (0x20..0x7e)
isspace(c) : 1 dacă c este : ' ', '\t', '\n', '\r', '\f' sau '\v'
isalpha(c) : echivalent cu isupper(c) | | islower(c)
isalnum(c) : echivalent cu isalpha(c) | | isdigit(c)
ispunct(c) : echivalent cu isgraph(c) && !isalnum(c)

Codurile numerice specificate mai sus corespund codului ASCII. Este recomandabil să se folosească aceste funcţii şi nu testarea explicită conform codului ASCII, deoarece aceasta din urmă va eşua pe maşini care folosesc alte coduri (de exemplu codul EBCDIC).

Funcţiile **int tolower(c);** şi **int toupper(c);** întorc c convertit din literă mare în mică şi reciproc sau c nemodificat dacă nu este literă.

EXERCIŢII

6. Scrieţi un program care să accepte un nume generic de fişier şi care să transforme fiecare fişier, convertind literele mici în litere mari şi lăsând celelalte caractere neschimbate. Conversia trebuie să se facă în acelaşi fişier.

7. Scrieţi un program care să accepte un nume de fişier (presupus a conţine text) şi care să afişeze o statistică de forma : număr de caractere, număr de linii, număr de cifre zecimale, litere mici şi mari etc.

22.4. FUNCȚII PENTRU OPERAȚII CU ȘIRURI DE CARACTERE : <string. h>

În funcțiile de mai jos, al căror nume începe cu „**str**", variabilele **s** și **t** sunt de tip **char** *, **cs** și **ct** sunt de tip **const char** *, **n** este de tip **size_t** și **c** este de tip **int**, convertit la **char**.

- **char *strcpy(s, ct)**

Copiază șirul **ct** în **s**, inclusiv '\0'. Întoarce **s**.

- **char *strncpy(s, ct, n)**

Copiază cel mult **n** caractere din **ct** în **s**, completând **s** cu '\0', dacă **ct** are mai puțin de **n** caractere. Dacă se copiază **n** caractere, nu se adaugă '\0'. Întoarce **s**.

- **char *strcat(s, ct)**

Concatenează șirul **ct** la sfârșitul lui **s**. Întoarce **s**.

- **char *strncat(s, ct, n)**

Concatenează cel mult **n** caractere din **ct** la sfârșitul lui **s**. Adaugă '\0' după concatenare. Întoarce **s**.

- **int strcmp(cs, ct)**

Compară **cs** și **ct** lexicografic. Întoarce < 0 dacă cs<ct, 0 dacă **cs == ct** sau > 0 dacă cs>ct.

- **int strncmp (cs, ct, n)**

Compară cel mult **n** caractere din **cs** și **ct**. Întoarce similar cu **strcmp()**.

- **char *strchr(cs, c)**

Întoarce pointerul la prima apariție a lui **c** în **cs** sau **NULL** dacă **c** nu apare în **cs**.

- **char *strrchr(cs, c)**

Întoarce pointerul la ultima apariție a lui **c** în **s** sau **NULL** dacă **c** nu apare în **cs**.

- **size_t strspn(cs, ct)**

Întoarce lungimea prefixului lui **cs**, care conține numai caractere din **ct** (poate fi și 0).

- **size_t strcspn(cs, ct)**

Întoarce lungimea prefixului lui **cs**, care conține numai caractere care nu sunt în **ct**.

- **char *strpbrk(cs, ct)**

Întoarce pointerul la prima apariție în **cs** a oricărui caracter din **ct** sau **NULL**, dacă în **cs** nu apare nici un caracter din **ct**.

- **char *strstr(cs, ct)**

Întoarce pointerul la prima apariție a lui **ct** ca subșir al lui **cs** sau **NULL**, dacă **ct** nu este subșir al lui **cs**.

- **size_t strlen(cs)**

Întoarce lungimea lui **cs** (număr de caractere). '\0' nu se numără.

- **char *strtok(s, ct)**

Se caută în **s** subșiruri delimitate de caractere din **ct**. O secvență de apeluri ale funcției **strtok(s, ct)** va descompune **s** în subșiruri delimitate de oricare caracter din **ct**. Primul apel trebuie făcut cu **s** diferit de **NULL**. Se va întoarce pointerul la primul subșir din **s**, format cu caractere care nu sunt în **ct**. Primul caracter de după acest subșir este înlocuit cu '\0'. Fiecare din apelurile următoare, care trebuie făcute punând **NULL** pe post de **s**, va întoarce următorul subșir, în aceeași manieră. **strtok** va întoarce **NULL** când nu se mai detectează astfel de subșiruri. Șirul **ct** poate fi diferit de la un apel la altul.

- **char *strerror(n)**

Întoarce un pointer către un șir fix, depinzând de implementare, care descrie eroarea cu codul **n**. Se poate folosi variabila externă **errno**.

Funcţiile următoare (care încep cu „mem") sunt destinate operaţiilor cu diverse obiecte din memorie, văzute ca tablouri de caractere ('\0' nu mai are rol de termintor). Parametrii **s** şi **t** sunt de tip **void ***, **cs** şi **ct** sunt de tip **const void ***, **n** este de tip **size_t**, iar **c** este de tip **int** convertit la **unsigned char**.

- **void *memcpy(s, ct, n)**

Copiază **n** caractere de la **ct** în **cs**. Întoarce **s**.

- **void *memmove(s, ct, n)**

La fel ca **memcpy**, dar funcţionează corect şi dacă s şi **ct** se întrepătrund.

- **int memcmp(cs, ct, n)**

Compară primii **n** octeţi din **cs** şi **ct**. Întoarce un întreg calculat la fel ca la **strcmp()**.

- **void *memchr(cs, c, n)**

Întoarce pointerul la prima apariţie a lui **c** în **cs** sau **NULL**, dacă **c** nu este în primele **n** caractere din **cs**.

- **void *memset"(s, c, n)**

Pune caracterul **c** în primele **n** caractere din **s**. Întoarce **s**.

Cu excepţia funcţiei **memmove**, toate funcţiile de copiere (atât cele cu numele „**str**.." cât şi cele cu numele „**mem**...", vor conduce la efecte nedefinite dacă zonele sursă şi destinaţie se întrepătrund.

Exemplu. Funcţia **memmove** poate fi folosită la o implementare eficientă a unei funcţii **swap** universale, care schimbă două obiecte de orice tip din memorie.

```
void swap (void *a, void *b, size_t n)
{
      void *temp;
      if ((temp=malloc(n))==NULL) {
            fprintf (stderr, "Swap: eroare alocare...\n"); exit (1);
      }
      memmove (temp, a, n);
      memmove (a, b, n);
      memmove (b, temp, n);
      free (temp);
}
```

EXERCIŢII

8. Scrieţi un program care să citească linii de text de la consolă până când se introduce ⟨CR⟩ pe o linie vidă şi apoi să afişeze cuvintele introduse, câte unul pe o linie. Cuvintele se consideră delimitate de caractere ' ','\t' şi '\n'. Despărţirea în cuvinte se face cu funcţia **strtok**.

*__9.__ Scrieţi un program care, primind în linia de comandă un nume de fişier text şi un şir de caractere care nu conţine spaţii albe, afişează numărul de apariţii al şirului dat în fişierul dat.

*__10.__ Scrieţi un program care, primind un nume de fişier text şi două şiruri de caractere care nu conţin spaţii albe, creează un alt fişier (eventual acelaşi nume, dar altă extensie) în care fiecare apariţie a primului şir să fie înlocuită cu al doilea şir.

22.5. FUNCŢII MATEMATICE : ⟨math. h⟩

Apelurile de funcţii matematice (care sunt funcţii de tip **double**, cu argument(e) **double**) pot conduce la erori de execuţie. Valorile întregi predefinite **EDOM** şi **ERANGE** (definite în ⟨**errno.h**⟩) semnalează eroare de domeniu şi respectiv, de reprezentare. **Eroare de domeniu** înseamnă că *un argument*

al unei funcții matematice este în afara domeniului admisibil (de exemplu valori negative la funcția radical), caz în care **errno** se poziționează la valoarea **EDOM**. Valoarea întoarsă de funcție în acest caz depinde de implementare, dar unele funcții întorc valoarea double **HUGE_VAL** (o valoare „mare"). **Eroare de reprezentare** înseamnă că *rezultatul unei funcții nu se poate reprezenta corect ca* **double**. Dacă este vorba de depășire superioară, se întoarce **HUGE_VAL**, cu semnul corect, iar **errno** este poziționat la **ERANGE**. Dacă este vorba de depășire inferioară, funcția întoarce 0, caz în care poziționarea lui **errno** la **ERANGE** depinde de implementare.

În descrierea următoare, argumentele **x** și **y** sunt de tip double, **n** este întreg și toate funcțiile întorc tipul **double**. La funcțiile trigonometrice, unghiurile se consideră în radiani.

sin(x)	sinus de x
cos(x)	cosinus de x
tan(x)	tangenta de x.
asin(x)	arcsinus de x, în domeniul $[-\pi/2, \pi/2]$, $x\in[-1, 1]$
acos(x)	arccosinus de x, în domeniul $[0, \pi]$, $x\in[-1, 1]$
atan(x)	arctangentă de x, în domeniul $(-\pi/2, \pi/2)$
atan2(y,x)	argumentul lui $x+iy$, în domeniul $(-\pi, \pi]$
sinh(x)	sinus hiperbolic de x
cosh(x)	cosinus hiperbolic de x
tanh(x)	tangenta hiperbolică de x
exp(x)	exponențială de x
log(x)	logaritm natural de x, $x>0$
log10(x)	logaritm zecimal de x, $x>0$
pow(x, y)	x^y. Apare eroare de domeniu dacă $x=0$ și $y\leq 0$ sau dacă $x<0$ și y nu este întreg
sqrt(x)	rădăcină pătrată din x, $x\geq 0$
ceil(x)	cel mai mic întreg $\geq x$, întors ca double
floor(x)	cel mai mare întreg $\leq x$, întors ca double
fabs(x)	valoarea absolută a lui x
ldexp(x,n)	$x*2^n$
frexp(x,int *exp)	desparte x în mantisă normalizată în intervalul $[1/2, 1)$, valoare care este întoarsă de funcție și o putere a lui 2 care este depusă în *exp. Dacă x este 0, atunci ambele valori vor fi 0.
modf(x, double *ip)	desparte x în partea întreagă care se depune în *ip și partea fracționară care se întoarce. Ambele au același semn ca x.
fmod(x,y)	restul împărțirii lui x la y (ca întregi), întors ca valoare double, cu același semn ca x. Dacă x este 0, rezultatul depinde de implementare.

22.6. FUNCȚII UTILITARE : ⟨stdlib. h⟩

● **double atof(const char *s);**
Convertește s la **double**. Echivalent cu: **strtod(s, NULL).**
● **int atoi(const char *s);**
Convertește s la **int.** Echivalent cu **(int) strtol(s, NULL, 10).**
● **long atol(const char *s);**
Convertește s la **long.** Echivalent cu **strtol(s, NULL, 10).**

● **double strtod(const char *s, char **endp);**

Convertește s la **double**, ignorând spațiile albe inițiale. Conversia se oprește la primul caracter incorect ca număr. Dacă **endp** este != NULL, atunci depune în ***endp** pointerul la primul caracter neconvertit. Evident, **endp** poate avea și valoarea **NULL**. Dacă apare depășire superioară, se întoarce **HUGE_VAL**, cu semnul respectiv. Dacă apare depășire **inferioară**, se întoarce 0. În ambele cazuri, errno se poziționează la **ERANGE**.

● **long strtol(const char *s, char **endp, int radix);**

Convertește s la **long**, ignorând spațiile albe inițiale. Conversia se oprește la primul caracter incorect ca număr. Dacă **endp** este != **NULL**, atunci depune în ***endp** pointerul la primul caracter neconvertit. Evident **endp** poate avea și valoarea **NULL**. Dacă **radix** este între **2** și **36**, conversia se face presupunând că s este scris în baza **radix**. Dacă **radix** este **0**, se consideră baza **8**, **10** sau **16**. O cifră inițială **0** va împiedica baza **8**, iar **0x** sau **0X** baza **16**. Literele mici sau mari vor reprezenta cifrele cuprinse între **10** și **radix-1**. Dacă **radix** este **16**, sunt permise prefixele **0x** sau **0X**. Dacă apare depășire superioară, se întoarce **LONG_MAX** sau **LONG_MIN** după cum este semnul rezultatului, iar **errno** este poziționat la **ERANGE**. De exemplu, apelând funcția cu **radix** = =**19**, cifrele permise vor fi: **'0'...'9'** și **'a'...'i'** sau **'A'...'I'**.

● **unsigned long strtoul(const char *s, char **endp, int radix);**

Similar cu **strtol**, exceptând tipul **unsigned long** și valoarea întoarsă în caz de depășire (superioară), care este **ULONG_MAX**.

● **int rand(void);**

Întoarce o valoare întreagă pseudo-aleatoare în domeniul **0... ...RAND_MAX**, care este cel puțin **32767**.

● **int srand(unsigned int seed);**

Inițializează generatorul de numere aleatoare cu valoarea **seed**.

● **void *malloc(size_t size);**

Întoarce un pointer către o zonă compactă din memoria dinamică, de dimensiune **size** octeți, sau **NULL**, dacă nu există spațiu disponibil.

● **void *calloc(size_t n, size_t size);**

Întoarce un pointer către o zonă compactă capabilă să memoreze un tablou de **n** obiecte, toate de dimensiune **size**, sau **NULL**, dacă nu există spațiu disponibil. Spațiul rezervat se inițializează cu 0.

● **void *realloc (void *p, size_t size) ;**

Schimbă lungimea zonei de memorie dinamică indicată de **p** la **size** octeți. Conținutul zonei de lungime minimă dintre dimensiunea veche și cea nouă va fi nemodificat. Dacă dimensiunea nouă este mai mare decât cea veche, noul spațiu este neinițializat. Se întoarce un pointer la zona realocată sau **NULL** dacă nu se poate satisface cererea, caz în care **p** este nemodificat. Zona indicată inițial trebuie să fi fost obținută prin **malloc** sau **calloc**.

● **void free(void *p);**

Eliberează zona indicată de **p**, care trebuie să fi fost obținută prin **malloc**, **calloc** sau **realloc**.

● **void exit(int status);**

Terminare normală a programului. Sunt apelate, în ordinea inversă în care au fost înregistrate, funcțiile marcate prin **atexit()**. Se scriu bufferele nescrise în fișiere, se închid toate fișierele deschise și se întoarce controlul mediului care a lansat programul în execuție, întorcându-se valoarea **status**. Se pot folosi valorile **EXIT_SUCCESS** sau **EXIT_FAILURE**. Valoarea **0** corespunde lui **EXIT_SUCCESS**, iar **1** lui **EXIT_FAILURE**.

• **int atexit(void (*func)(void));**

Marchează *func pentru a fi apelată la terminarea normală a programului. Întoarce ceva !=0, dacă nu se poate face marcarea.

• **int system(const char *s);**

Se transmite s mediului care a lansat programul (uzual interpretorul de comenzi al sistemului de operare), pentru execuție. Dacă s este **NULL**, se întoarce !=**0**, dacă există un interpretor de comenzi. Dacă s nu este **NULL**, valoarea întoarsă depinde de implementare. De remarcat că, în mediile de dezvoltare integrată, această funcție nu se poate folosi. Un program care folosește această funcție trebuie lansat în execuție din sistemul de operare.

• **char *getenv(const char *name);**

Întoarce un șir care conține o descriere a mediului asociat cu **name** sau **NULL**, dacă nu există un asemenea șir. Detaliile depind de implementare.

• **void *bsearch(const void *key, const void *base, size_t n, size_t size, PFCMP cmp);**

Funcția caută în tabloul ordonat **base[0]...base[n-1]** un obiect care să corespundă lui ***key**. Elementele tabloului au fiecare **size** octeți. **cmp** este un pointer la funcția de comparație, care este de tipul **PFCMP**, definit prin:

typedef int (*PECMP)(const void *, const void *);

Funcția ***cmp** trebuie să întoarcă ceva negativ dacă primul argument (care indică totdeauna cheia căutată) este mai mic decât al doilea (un element al tabloului), zero dacă este egal și pozitiv dacă este mai mare. Obiectele din tablou trebuie să fie în ordine crescătoare, conform funcției de comparație. Se întoarce un pointer la un obiect din tablou care corespunde lui ***key** sau **NULL**, dacă nu s-a găsit (o implementare a acestei funcții este prezentată în capitolul 20).

• **void qsort(void *base, size_t n, size_t size, PFCMP cmp);**

Sortează în ordine crescătoare tabloul **base[0]...base[n-1]**, cu elemente de dimensiune **size**, conform funcției de comparare ***cmp**, care primește adresele a două elemente ale tabloului întorcând un întreg la fel ca la **bsearch**.

• **int abs(int i);**

Întoarce un întreg egal cu valoarea absolută a lui **i**.

• **long labs(long l);**

Întoarce un întreg lung egal cu valoarea absolută a lui **l**.

• **div_t div(int num, int denom);**

Calculează câtul și restul împărțirii lui **num** la **denom** și le stochează în cei soi membri **quot** și **rem** ai structurii predefinite de tip **div_t**. Membrii structurii sunt de tip **int**. Se întoarce structura de tip **div_t**.

• **ldiv_t ldiv(long num, long denom);**

Calculează câtul și restul împărțirii lui **num** la **denom** și le stochează în cei doi membri **quot** și **rem** ai structurii predefinite de tip **ldiv_t**. Membrii structurii sunt de tip **long**. Se întoarce structura de tip **ldiv_t**.

EXERCIȚII

11. Folosind funcția **qsort**, scrieți un program care să accepte în linia de comandă numele a două fișiere. Primul fișier trebuie să existe și să conțină valori întregi în format zecimal extern. Programul trebuie să citească cel mult 1000 de întregi din primul fișier, să-i sorteze în ordine crescătoare și să-i scrie în al doilea fișier.

12. Scrieți o funcție **conv_rad** care primește două șiruri de caractere și doi întregi specificând două baze de numerație cuprinse între 2 și 36. Funcția trebuie să convertească primul

şir (interpretat ca un număr scris în prima bază) în al doilea şir, reprezentând aceeaşi valoare numerică, dar scrisă în a doua bază. De exemplu, un apel :

conv_rad ("1234", s, 10, 16);

va face ca în s să fie depusă reprezentarea lui 1234 în baza 16.

22.7. MACROINSTRUCŢIUNI PENTRU FUNCŢII CU NUMĂR VARIABIL DE ARGUMENTE : ⟨stdarg. h⟩

Funcţiile C pot fi apelate cu un număr variabil de parametri actuali, exemplul standard fiind funcţiile **printf** şi **scanf.** Macroinstrucţiunile din ⟨**stdarg.h**⟩ dau posibilitatea de a crea funcţii utilizator cu număr variabil de argumente. Aceste macroinstrucţiuni sunt **va_start, va_arg** şi **va_end.** Se foloseşte tipul predefinit **va_list,** pentru a accesa argumentele în număr variabil.

Prototipul unei funcţii cu număr variabil de parametri este de forma:

● **tip nume(lista de parametri,...);**

unde lista de parametri trebuie să fie nevidă, deci să existe cel puţin un parametru fix.

La apelul unei astfel de funcţii, numărul de argumente trebuie să fie mai mare sau egal cu numărul de parametri ficşi. Argumentele care corespund unor parametri în număr variabil suferă conversii implicite de tip astfel: toate argumentele de tip integral sunt promovate la tipul **int** şi toate argumentele de tip **float** sunt convertite la **double.**

Accesul din interiorul funcţiei la parametrii în număr variabil se face în felul următor. Fie **lastarg** ultimul parametru fix. Atunci, se declară în funcţie o variabilă de tip **va_list** (care va fi folosită ca pointer către următorul argument):

● **va_list pa;**

Variabila **pa** se iniţializează o singură dată prin apelul de **macro:**

va_start(pa, lastarg);

Folosind apoi repetitiv macrocomanda **va_arg,** după prototipul:

● **tip va_arg(va_list pa, tip);**

transmiţându-i ca parametru variabila **pa** şi un tip de date, se va întoarce următorul argument variabil, cu tipul **tip,** modificând corespunzător variabila **pa,** astfel încât să indice următorul argument. Se observă că nu se dispune de un mecanism prin care să se oprească acest proces (practic, **va_arg** extrage din stivă următorul parametru cu tipul **tip,** iar **va_start** poziţionează pointerul **pa** în stivă, după ultimul argument fix, adică pe primul argument variabil). Oprirea procesului repetitiv trebuie făcută pe baza unor informaţii extrase din parametrii ficşi. De exemplu, la **printf,** primul parametru, care se află în vârful stivei, este adresa şirului de caractere care descrie formatul. Acest şir este prezent întotdeauna, deci **printf** poate avea acces la el („ştie" unde se află). Pe baza informaţiilor din format, se extrag cu **va_arg** parametri variabili ai funcţiei (se ştie câţi sunt şi de ce tip este fiecare). Macroinstrucţiunea:

● **void va_end(va_list pa);**

trebuie apelată o singură dată după ce s-a extras ultimul parametru variabil, dar înainte de terminarea funcţiei.

Uneori, nu se extrag argumentele cu **va_arg,** ci numai se poziţionează variabila de tip **va_list** (cu **va_start**) şi se transmite această variabilă la o altă funcţie, care are parametru de tip **va_list.** Exemple tipice de astfel de funcţii sunt **vprintf, vsprintf** şi **vfprintf.**

Iată un exemplu de funcție, *specifică IBM-PC*, care scrie la consolă cu conversie de format (analog cu printf), dar *folosind funcții de BIOS :*

```
#include <dos. h>
#include <ctype. h>
#include <stdarg. h>
#define VIDEO 0x10
void bputch (char);                          /*Prototipuri*/
void bputs (char *);                         /*    de    */
int bprintf (char *fmt,...);                 /*  funcții  */
int bprintf (char *fmt,...)
{
    va_list argptr;
    int cnt;
    char buffer [120];
    va_start (argptr, fmt);
    cnt=vsprintf (buffer, fmt, argptr);
    va_end (argptr);
    bputs (buffer);
    return cnt;
}
void bputs (char *s)
{
    while (*s!='\0')
        bputch (*s++);
}
void bputch (char c)
{
    union REGS regs;
    unsigned char col, row;
    regs. h. ah=3;                           /*citește poziție*/
    regs. h. bh=0;                           /*cursor (pagina 0)*/
    int 86 (VIDEO, &regs, &regs);
    col=regs. h. dl; row=regs. h. dh;
    if (isprint (c)) {
        regs. h. ah=9; regs. h. al=c;
        regs. h. bh=0 regs. h. bl=7;   /*atribut normal*/
        regs. x. cx=1;                  /*1 caracter*/
        int86 (VIDEO, &regs, &regs);
        col++; col%=80;
        if (col==0) {
            row++; row%=25;
        }
    }
    else
    switch (c) {
        case '\r': col=0: break;
        case '\n': col=0; row++; row%=25; break;
        case '\t': if (col<72)
                        col+=8—col%8;
              break;
        case '\b': col=(col)? col—1:79;
              if (col==79)
```

```
                    row=(row)? row—1:24;
                break;
            default:    break;
        regs. h. ah=2; regs. h. bh=0;        /*seteaza poziţie*/
        regs. h. dh=row;                     /*  cursor      */
        regs. h. dl=col;
        int86 (VIDEO, &regs,  &regs);
        }
```

Funcţia **bprintf()** foloseşte tehnica enunţată mai sus pentru accesul la parametrii variabili, apelând apoi **vsprintf()**, care converteşte variabilele precizate de **argptr** în şirul **buf**, întorcând o valoare întreagă, la fel ca **printf()**. Acest şir este afişat la consolă cu **bputs()**, care foloseşte, la nivel de caracter, **bputch()**.

Funcţia **bputch(c)** citeşte şi memorează poziţia curentă a cursorului pe ecran cu funcţia BIOS corespunzătoare. Dacă **c** este caracter tipăribil, este afişat şi apoi se actualizează linia şi coloana (funcţia BIOS 0x10, subfuncţia 9 nu schimbă poziţia cursorului). Linia şi coloana se incrementează % 25 şi, respectiv, % 80, deci corespunzător unei afişări în mod pagina (fără **scroll**). Dacă **c** nu este tipăribil, se tratează câteva cazuri uzuale de caractere de control: CR, LF, TAB şi BS. Aceste cazuri ar putea fi extinse cu '\a' (alarmă sonoră), '\v' (tabulare verticală) etc. Este posibil să se realizeze şi operaţii mai complexe cu ecranul, cum ar fi **scroll-up** sau **scroll-down**.

Funcţia **bprintf()** se apelează acum exact ca şi **printf()**. O situaţie similară se întâlneşte când se lucrează în mediu grafic, unde de obicei există funcţii de tipărire la nivel grafic numai pentru şiruri de caractere. Cu această tehnică se poate construi o funcţie **gprintf()** de tipărire cu format, în mod grafic.

Apelul funcţiei **int86** este specific Borland C, iar uniunea de tip **REGS** este folosită pentru a citi/scrie din/în registrele 8086.

EXERCIŢII

13. Scrieţi o funcţie **eprintf** cu număr variabil de argumente, similară cu **printf**, care să tipărească un mesaj de eroare şi apoi argumentele primite, exact ca **printf**.

*__14.__ Scrieţi o mică funcţie **mprintf**, cu număr variabil de argumente, care să trateze, exact ca **printf**, specificatorii de format **%d**, **%s**, **%c** şi secvenţele escape uzuale (**\n**, **\t**, ****, **\'**, **\"** etc.). Folosiţi o funcţie proprie pentru conversie de la întreg la şir de caractere ASCII şi apoi afişaţi şirul obţinut.

22.8. FUNCŢII PENTRU GESTIUNEA TIMPULUI (DATĂ ŞI ORĂ) : ⟨time. h⟩

În acest header sunt declarate diverse funcţii pentru dată şi oră. Unele funcţii procesează timpul local, care poate diferi de timpul astronomic, din cauza zonelor de timp, oră de vară, etc. Tipurile predefinite **clock_t** şi **time_t** sunt tipuri aritmetice adecvate procesării timpului. Structura de tip **struct tm**, predefinită, conţine următorii membri:

int tm_sec;	secunda	(0. . .59)
int tm_min ;	minut	(0. . .59)
int tm_hour;	ora	(0. . .23)
int tm_mday;	ziua din lună	(1. . .31)
int tm_mon;	luna	(0. . .11)
int tm_year;	an	(de la 1900)
int tm_wday;	ziua săptămânii	(0. . .6)
int tm_yday;	ziua din an	(0. . .365)
int tm_isdst;	flag pentru ora de vară	

Flagul **tm_isdst** este 1 dacă este ora de vară, 0 dacă nu, şi —1 dacă această informaţie nu este disponibilă.

● **clock_t clock(void);**

Întoarce timpul U.C. (numărul de tacte de ceas de timp real), scurs de la începerea programului, sau —1 dacă nu este disponibil. Valoarea reală **clock()/CLK_TCK** reprezintă acest timp în secunde.

● **time_t time(time_t *tp);**

Întoarce timpul de calendar sau —1 dacă nu este disponibil. Aceeaşi valoare este depusă în ***tp**, dacă **tp** nu este **NULL**. Timpul de calendar poate fi, de exemplu, numărul de secunde de la 1 ianuarie 1970, ora 00:00:00 (la Borland C).

● **double diff_time(time_t time2, time_t time1);**

Întoarce **time2—time1**, exprimat în secunde.

● **time_t mktime(struct tm *tp);**

Converteşte timpul local din structura ***tp** în timp de calendar, în aceeaşi reprezentare ca şi **time()**. Întoarce timpul de calendar sau —1, dacă nu poate fi exprimat corect.

Următoarele 4 funcţii întorc pointeri către obiecte statice din memorie, care pot fi suprascrise de alte funcţii.

● **char *asctime(const struct tm *tp);**

Converteşte timpul din structura ***tp** într-un şir de caractere de forma:

Sat Oct 31 12:10:50 1992\n\0

● **char *ctime(const time_t *tp);**

Converteşte timpul de calendar ***tp** într-un şir de caractere reprezentând timpul local. Echivalent cu: **asctime(localtime(tp)).**

● **struct tm *gmtime(const time_t *tp);**

Converteşte timpul de calendar ***tp** într-o structură de tip **tm** care va conţine Timpul Coordonat Universal (Greenwich Meridan Time). Întoarce **NULL** dacă nu este disponibil.

● **struct tm *localtime(const time_t *tp);**

Converteşte timpul de calendar ***tp** în timp local, întorcând un pointer la o structură de tip **tm.**

● **size_t strftime(char *s, size_t smax, const char *fmt, const struct tm *tp);**

Converteşte datele din structura ***tp** în şirul s, conform cu formatul **fmt.** Formatul e similar cu cel de la **printf()**, în sensul că se pun în s caracterele obişnuite din **fmt**, iar cele care încep cu % sunt considerate descriptori de conversie. Se scriu cel mult **smax** caractere în s (inclusiv terminatorul). Funcţia întoarce numărul de caractere scrise în s (fără '\0'), sau 0 dacă s-au produs mai mult de **smax** caractere. Descriptorii de format sunt:

%**a**	numele zilei abreviat (de ex. Sat)
%**A**	numele zilei complet (de ex. Saturday)
%**b**	numele lunii abreviat (de ex. Oct)
%**B**	numele lunii complet (de ex. October)
%**c**	reprezentare locală a datei şi orei
%**d**	ziua din luna (01...31)
%**H**	ora (ceas cu 24 de ore) (00...23)
%**I**	ora (ceas cu 12 ore) (01...12)
%**j**	ziua din an (001...366)

%m luna din an (01...12)
%M minutul (00...59)
%p AM sau PM (sau echivalente locale)
%S secunda (00...59)
%U numărul săptămânii din an (considerând Luni prima zi a săptămânii) (00...53)
%W numărul săptămânii din an (considerând Duminică prima zi a săptămânii) (00...53)
%x reprezentare locală a datei
%X reprezentare locală a orei
%y anul (fără secol) (00...99) (de ex. 92)
%Y anul (cu secol) (de ex. 1992)
%Z numele zonei orare (dacă există)
%% caracterul %

Să considerăm următorul exemplu de program :

```
#include <stdio. h>
#include <stdlib. h>
#include <time. h>
void main (void)
{
	struct tm *timp;
	time_t t;
	char s[200];
	time ( &t);
	timp=localtime ( &t);
	strftime (s, 200,
	"%a %d %b %y %I %M %S% p\n"
		"%A %d %B %Y %H %M %S\n"
		"%c\n"
		"Ziua săptămânii : %w Ziua anului: %j Luna: %m Săptămâna:
		%U\n"
		"Local: %x %X\n Zona: %Z\n", timp);
	puts(s);
}
```

În urma execuției, se va afișa un text de forma :

```
Sat 31 Oct 92 01 09 53 PM
Saturday 31 October 1992 13 09 53
Sat Oct 31 13:09:53 1992
Ziua săptămânii : 6 Ziua anului: 305 Luna: 10 Săptămâna: 43
Local: Sat Oct 31, 1992, 13:09:53
Zona: EST
```

EXERCIȚII

* **15.** Scrieți o funcție adecvată care, apelată la începutul și respectiv sfârșitul programului, să realizeze, la al doilea apel, afișarea duratei de execuție a programului respectiv. Cum se poate rezolva problema în cazul în care programul își încheie execuția din mai multe puncte posibile ?

16. Definiți o structură adecvată pentru a memora numele unei persoane și data sa de naștere. Definiți un tabel de astfel de structuri și folosiți-l intr-un program care să citească data curentă și să afișeze un mesaj adecvat la consolă, în situația în care data respectivă este data nașterii unei persoane din tabel. Verificați că programul funcționează corect și în situația în care în tabel există mai multe persoane cu aceeași zi de naștere.

22.9. LIMITE DEPENDENTE DE IMPLEMENTARE : ⟨limits. h⟩, ⟨float. h⟩

În aceste headere sunt definite constante care dau valorile minime și maxime pentru tipurile de bază. Se prezintă în continuare valorile folosite în implementarea Borland C:

CHAR_BIT	8	Număr de biți pe caracter
CHAR_MAX	UCHAR_MAX sau SCHAR_MAX	Val. max. pentru char
CHAR_MIN	0 sau SCHAR_MIN	Val. min. pentru char
INT_MAX	+32767	Val. max. pentru int
INT_MIN	−32768	Val. min. pentru int
LONG_MAX	+2147483647L	Val. max. pentru long
LONG_MIN	−2147483648L	Val. min. pentru long
SCHAR_MAX	+127	Val. max. pentru signed char
SCHAR_MIN	−128	Val. min. pentru signed char
SHRT_MAX	+32767	Val. max. pentru short
SHRT_MIN	−32768	Val. min. pentru short
UCHAR_MAX	255U	Val. max. pentru unsigned char
UINT_MAX	65535U	Val. max. pentru unsigned int
ULONG_MAX	4294967295UL	Val. max. pentru unsigned long
USHRT_MAX	65535U	Val. max. pentru unsigned short

Constantele următoare descriu implementarea tipului **float**:

FLT_RADIX	2	Baza față de care se consideră exponentul
FLT_ROUNDS	1	Modul de rotunjire la adunare
FLT_DIG	6	Numărul de cifre semnificative (**float**)
FLT_EPSILOR	1.19E−7	Precizia mașinii (**float**). Cel mai mic x a.î. 1.0f+x!=1.0f
FLT_MANT_DIG	24	Numărul de cifre în baza FLT_RADIX ale mantisei
FLT_MAX	3.40E+38	Cel mai mare **float** pozitiv
FLT_MIN	1.17E−38	Cel mai mic **float** pozitiv, normalizat
FLT_MAX_EXP	+128	Cel mai mare n a.î. FLT_RADIX^{n} −1 este reprezentabil ca **float**
FLT_MIN_EXP	−125	Cel mai mic n a.î. FLT_RADIX^{n} este un **float** normalizat
FLT_MAX_10_EXP	+38	Cel mai mare n a.î. 10^{n} este reprezentabil ca **float**
FLT_MIN_10_EXP	−37	Cel mai mic n a.î. 10^{n} este un **float** normalizat

Constantele următoare descriu implementarea tipului **double**:

DBL_DIG	15	Numărul de cifre semnificative (**double**)
DBL_EPSILON	2.22E−16	Precizia mașinii (**double**). Cel mai mic x a.î. 1.0+x!=1.0
DBL_MANT_DIG	53	Numărul de cifre în baza FLT-RADIX ale mantisei
DBL_MAX	1.79E+308	Cel mai mare **double** pozitiv
DBL_MIN	2.22E−308	Cel mai mic **double** pozitiv, normalizat

DBL_MAX_EXP	+1024	Cel mai mare n a.î. $FLT_RADIX^n - 1$ este reprezentabil ca **double**
DBL_MIN_EXP	−1021	Cel mai mic n a.î. FLT_RADIX^n este un **double** normalizat
DBL_MAX_10_EXP	+308	Cel mai mare n a.î. 10^n este reprezentabil ca **double**
DBL_MIN_10_EXP	−307	Cel mai mic n a.î. 10^n este un **double** normalizat

Constantele următoare descriu implementarea tipului long **double**:

LDBL_DIG	19	Numărul de cifre semnificative long **double**)
LDBL_EPSILON	1.08E−19	Precizia mașinii (long **double**). Cel mai mic x a.î. 1.0L+x! = 1.0L
LDBL_MANT_DIG	64	Numărul de cifre în baza FLT_RADIX ale mantisei
LDBL_MAX	1.18E+4932	Cel mai mare long **double** pozitiv
LDBL_MIN	1.00E−4932	Cel mai mic long **double** pozitiv, normalizat
LDBL_MAX_EXP	+16384	Cel mai mare n a.î. $FLT_RADIX^n - 1$ este reprezentabil ca long **double**
LDBL_MIN_EXP	−16381	Cel mai mic n a.î. FLT_RADIX^n este un long **double** normalizat
LDBL_MAX_10_EXP	+4932	Cel mai mare n a.î. 10^n este reprezentabil ca long **double**
LDBL_MIN_10_EXP	−4931	Cel mai mic n a.î. 10 este un long **double** normalizat

Reprezentarea internă a numerelor reale (formatul standard IEEE pentru **float** și **double** și formatul intern Intel pe 10 octeți pentru **long double**) este următoarea:

a) **float**

$$x = (-1)^s * 1.f_{22}f_{21}\ldots f_1f_0 * 2^{e-127}$$

s este bitul de semn, $e = e_7e_6\ldots e_1e_0$ este exponentul deplasat cu 127, iar $f = f_{22}f_{21}\ldots f_1f_0$ este mantisa (fractia) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 23 al mantisei) nu se reprezintă intern. Cei patru octeți sunt dați în tabelul 22.1.

Tabelul 22.1

adr. low	f7	f6	f5	f4	f3	f2	f1	f0
	f15	f14	f13	f12	f11	f10	f9	f8
	e0	f22	f21	f20	f19	f18	f17	f16
adr. high	s	e7	e6	e5	e4	e3	e2	e1

b) double

$x = (-1)^s * 1.f_{51}f_{50}...f_1f_0 * 2^{e-1023}$

s este bitul de semn, $e = e_{10}e_9...e_1e_0$ este exponentul deplasat cu 1023, iar $f = f_{51}f_{50}...f_1f_0$ este mantisa (fracția) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 52 al mantisei) nu se reprezintă intern. Cei opt octeți sunt dați în tabelul 22.2

Tabelul 22.2.

adr. low	f7	f6	f5	f4	f3	f2	f1	f0
	f15	f14	f13	f12	f11	f10	f9	f8
	f23	. . . .						f16
	f31	. . . .						f24
	f39	. . . .						f32
	f47	. . . .						f40
	e3	e2	e1	e0	f51	f50	f49	f48
adr. high	s	e10	e9	e8	e7	e6	e5	e4

c) long double

$x = (-1)^s * 1.f_{62}f_{61}...f_1f_0 * 2^{e-16383}$

s este bitul de semn, $e = e_{14}e_{13}...e_1e_0$ este exponentul deplasat cu 16383, iar $f = f_{62}f_{61}...f_1f_0$ este mantisa (fracția) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 63 al mantisei) se reprezintă intern ca un bit totdeauna egal cu 1, cu excepția valorii long double 0,0L, când acest bit este 0. Cei zece octeți sunt dați în tabelul 22.3.

Tabelul 22.3.

adr. low	f7	f6	f5	f4	f3	f2	f1	f0
	f15	f14	f13	f12	f11	f10	f9	f8
	f23	. . . .						f16
	f31	. . . .						f24
	f39	. . . .						f32
	f47	. . . .						f40
	f55	. . . .						f48
	1	f62	f61	f60	f59	f58	f57	f56
	e7	e6	e5	e4	e3	e2	e1	e0
adr. high	s	e14	e13	e12	e11	e10	e9	e8

Iată câteva exemple de reprezentare internă :

```
1.0f  =00 00 80 3f                      (s=0, f=0, e=  0x7f)
1.0   =00 00 00 00 00 00 f0 3f          (s=0, f=0, e= 0x3ff)
1.0Lf =00 00 00 00 00 00 00 80 ff 3f    (s=0, f=0, e=0x3fff)
```

```
2.0f   =00 00 00 40                      (s=0, f=0, e= 0x80)
2.0     00 00 00 00 00 00 00 40          (s=0, f=0, e= 0x400)
2.0Lf  =00 00 00 00 00 00 00 80 00 40    (s=0, f=0, e=0x4000)
0.5f   =00 00 00 3f                      (s=0, f=0, e= 0x7e)
0.5    =00 00 00 00 00 00 e0 3f          (s=0, f=0, e= 0x3fe)
0.5Lf  =00 00 00 00 00 00 00 80 fe 3f    (s=0, f=0, e=0x3ffe)
-1.0f  =00 00 80 bf                      (s=1, f=0, e= 0x7f)
-1.0   =00 00 00 00 00 00 f0 bf          (s=1, f=0, e= 0x3ff)
-1.0Lf =00 00 00 00 00 00 00 80 ff bf    (s=1, f=0, e=0x3fff)
```

Programul următor citeşte de la consolă valori long double şi afişează cele trei tipuri reale (**float, double** şi **long double**), în hexazecimal. Dacă se introduce 0, programul se termină. De observat că programul nu depinde de implementarea celor trei tipuri de numere reale, datorită folosirii operatorului **sizeof**.

```
#include <stdio.h>

typedef unsigned char BYTE;

void main(void)
{
    int i;
    float f; double d; long double ld;
    BYTE *p;

    do {
        scanf ("%Lf", &ld);
            f=(float) ld;
            d=(double) ld;

        for (i=0, p=(BYTE *) &f; i < sizeof(float); i++)
            printf ("%02x", *p++);
        putchar ('\n');
        for (i=0, p=(BYTE *) &d; i < sizeof(double); i++)
            printf ("%02x", *p++);
        putchar ('\n');
        for (i = 0, p = (BYTE *)&ld; i<sizeof ('ong double); i++)
            printf ("%02x", *p++);
        putchar ('\n');
    } while (!(ld==0.0L));
}
```

În implementarea **Borland Pascal (Turbo Pascal)**, pe lângă tipurile standard reale (4, 8 şi 10 octeţi), mai există un tip nestandard, numit **REAL**, moştenit de la versiuni mai vechi. Acest tip este implementat în Borland Pascal pe 6 octeţi, cu 39 de biţi pentru mantisă (normalizată) şi 8 biţi pentru exponent. Exponentul este deplasat cu 129. Reprezentarea internă este următoarea :

REAL (Borland Pascal)

$$x = (-1)^s * 1.f_{38}f_{37}...f_1f_0 * 2^{e-129}$$

s este bitul de semn, $e = e_7e_6...e_1e_0$ este exponentul deplasat cu 129, iar $f = f_{38}f_{37}...f_1f_0$ este mantisa (fracţia) normalizată. Bitul cu valoarea 1 din formula de mai sus (bitul 39 al mantisei) nu se reprezintă intern. Cei şase octeţi sunt daţi în tabelul 22.4.

Tabelul 22.4

adr. low	f7	f6	f5	f4	f3	f2	f1	f0
	f15	f14	f13	f12	f11	f10	f9	f8
	f23	f22	f21	f20	f19	f18	f17	f16
	f31	.	.	.	.	.	.	f24
	e0	f38	f37	f36	f35	f34	f33	f32
adr. high	s	e7	e6	e5	e4	e3	e2	e1

Dacă un program **PASCAL** generează date **REAL** pe un suport extern (disc), date care trebuie prelucrate apoi de un program **C**, se pune problema conversiei de la tipul **REAL** din **PASCAL** la unul din tipurile **float, double** sau **long double** existente în **C**. De asemenea, se pune şi problema conversiei inverse (de la tipurile reale din **C** la tipul **REAL** din **PASCAL**).

Următoarele două funcţii realizează conversiile dintre tipurile **REAL** (**PASCAL**) şi **double** (**C**).

```
typedef unsigned char BYTE;
typedef unsigned short WORD;
typedef BYTE REAL[6];
typedef union { BYTE c[8]; double d; } UD;

double rtod (REAL r)
{
/*
        Converteşte REAL la double
*/
        UD u;
        WORD le;

        if((le=r[0])!=0)
              le+=1023—129;

        u.c[7]=r[5] &0×80;              /* bitul de semn */

        u.c[7];= (le >> 4) & 0×ff;
        u.c[6] = (le << 4) & 0×ff;
        u.c[6] ;= (r[5] & 0×7f) >> 3;
        u.c[5] = ((r[5] & 0×7) << 5) ; (r[4] >> 3) ;
        u.c[4] = ((r[4] & 0×7) << 5) ; (r[3] >> 3) ;
        u.c[3] = ((r[3] & 0×7) << 5) ; (r[2] >> 3) ;
        u.c[2] = ((r[2) & 0×7) << 5) ; (r[1] >> 3) ;
        u.c[1] = u.c[0] = 0 ;

        return u.d;
}
```

```
void dtor(double x, REAL r)
{
/*
      Converteşte double la REAL
/*
      WORD e;
      UD u;

      u.d=x;
      e = ((WORD) (u.c[7]  & 0 x 7f)) << 4 ;
      e; = ((WORD) (u.c[6]  & 0 x f0)) >> 4 ;

      r[0] = (e) ? (BYTE)  ((e— 1023 + 129)  & 0 x ff): 0;
      r[5] = (u.c[7]  & 0 x 80);        /* bitul de semn */

      r[5] ;= ((u.c[6]  & 0 x 0f) << 3) ;  ((u.c[5]  & 0 x e0) >> 5) ;
      r[4] = ((u.c[5]  & 0 x 1f) << 3) ;  ((u.c[4]  & 0 x e0) >> 5) ;
      r[3] = ((u.c[4]) & 0 x 1f) << 3) ;  ((u.c[3]  & 0 x e0) >> 5) ;
      r[2] = ((u.c[3]  & 0 x 1f) << 3) ;  ((u.c[2]  & 0 x e0) >> 5) ;
      r[1] = ((u.c[2]  & 0 x 1f) << 3) ;  ((u.c[1]  & 0 x e0) >> 5) ;
}
```

RĂSPUNSURI ŞI REZOLVĂRI

Capitolul 5

5. 0.004567 9.81 3.00 18.85 2827.43 93000
7. a) 'D' b) 'e'
9. a) corectă b) incorectă
10. TRUE dacă a este zero sau multiplu al lui b, FALSE în caz contrar
12. (c>='0') **and** (c<='9')
13. (x **mod** y=0)
14. (luna=12) **and** (ziua<=25)
15. (a **mod** 4=0) **and** (a **mod** 100 <> 0) **or** (a **mod** 400=0)

Capitolul 7

1.

```
program AfMedia;
var
        a, b: integer;
        media: real;
begin
        writeln ('Precizaţi 2 numere întregi, pozitive, strict mai mici ca 1000');
        write ('A=');
        readln(a);
        write ('B=');
        readln(b);
        media:=(a+b)/2;
        writeln ('A=', a:3, 'B=', b:3, 'MEDIA=', media:5:2);
        readln;
end.
```

2.

```
program Cerc;
var
        raza, lung, aria:real;
begin
        write ('Precizaţi raza cercului:');
        readln (raza);
        writeln;
        writeln ('Raza cercului este: ', raza:6:3);
        writeln ('Lungimea cercului:', 2*pi*raza:6:3);
        writeln ('Aria cercului este:', pi*raza*raza:9:4);
        readln;
end.
```

3.

```
program Transformare;
var
        xrad, xgr, min, sec, zec:real;
begin
        write ('Introduceţi x în radiani:');
        readln(xrad);
        xgr:=(xrad*180)/pi;
        min :=(xgr —trunc(xgr))*60;
        sec :=(min—trunc(min))*60;
        zec :=(sec—trunc(sec))*10;
        writeln ('Echivalentul în grade minute/secunde/zecimi de sec. este:');
        writeln ('grade      =', trunc(xgr));
        writeln('minute      =', trunc(min));
        writeln('secunde     =', trunc(sec));
        writeln ('zecimi     =', trunc(zec));
        readln
end.
```

4.

```
program Puteri;
var
        x:real;
begin
        write ('Precizaţi valoarea x:');
        readln(x);
        writeln ('Numărul x':24, 'Puterea a 2-a':24, 'Puterea a 3-a':24);
        writeln (x:24:2, x*x:24:2, x*x*x:24:2);
        readln
end.
```

5.

```
program Data;
var
        zi, luna, an:integer;
begin
        writeln ('Precizaţi data:');
        write ('Ziua:');
        readln (zi);
        write ('luna:');
        readln (luna);
        write ('Anul:');
        readln (an);
        an:=an mod 100;
        writeln ('Data în formatul cerut este:');
        writeln (zi, '/', luna, '/', an);
        readln;
end.
```

Capitolul 8

1. incorecte: a şi d.
2. c, d, f, i.
3. d şi g sunt erori sesizate la compilare, iar b şi h pot genera erori la execuţie.

5.

```
i:=MaxInt;
while i>=MaxInt—100 do
    begin
        writeln (i:6);
        i:=i—1
    end.
```

7. Contorul ciclului (j) este modificat în interiorul buclei, lucru nepermis în cazul instrucțiunii **for**.
8. 120 (considerând Produs de tip real)
12. 20.0
13.

```
program EcGrad2;
var
    a, b, c:real;
    delta, x1, x2:real;
    begin
    write ('Precizați coeficienții a, b și c:');
    readln (a, b, c);
    if (a=0) then
        writeln ('Ecuația este de gradul II')
    else
        begin
                delta:=b*b—4*a*c;
                if (delta <0) then
                        writeln ('Ecuația nu are soluții reale!')
                else
                        begin
                            x1:=(—b—sqrt(delta))/(2*a);
                            x2:=(—b+sqrt (delta))/(2*a);
                            writeln ('Rădăcinile ecuației sunt:');
                            writeln ('x1=', x1:6:2,' și 'x2=', x2:6:2)
                        end
                end;
        readln
    end.
```

16.

```
program ElemMax;
var
    n, elem, max, i:integer;
begin
    write ('Precizați dimensiunea listei:');
    readln (n);
    max:=0;
    for i:=1 to n do
        begin
            write ('Precizați elementul', i:2, ':');
            readln (elem);
            if elem >max then
            max:=elem;
        end;
```

```
   writeln ('Elementul maxim din listă este:', max:5);
   readln
end.
```

19.

```
program Calcul Radical;
var
   x, eps, an, vechi:real;
begin
     write ('Precizaţi x:');
     readln (x);
     if (x < 0) then
        writeln ('X < 0, nu se poate extrage radicalul!')
     else
        begin
           write ('Precizaţi aproximaţia Eps:');
           readln (eps);
           an:=1;
           repeat
              vechi:=an;
              an:=(an+x/an)/2
           until (abs (a—vechi) < eps);
           writeln ('Radicalul din', x:5:2,' este ', an:6:2)
        end
end.
```

Capitolul 9

2. a) roşu; b) 0; c) 'D'; d) 23; e) 'E'
3. b), c). Instrucţiunea a) este corectă numai în cazul în care variabila i primeşte valori în subdomeniul de definiţie al variabilei j.
4. a) Deoarece numărul valorilor precizate de tipul index (integer) este egal cu numărul valorilor întregi cuprinse în domeniul —MaxInt...0...MaxInt, compilatorul va considera că se doreşte rezervarea unui spaţiu de memorie prea mare şi deci că declaraţia este incorectă.
   b) Declaraţia nu este corectă deoarece nu este permisă definirea unui subdomeniu al tipului real.

5.

```
program inversare;
type Martrice=array[1..10] of integer;
var n, i, buf:integer;
    Mat:Matrice;
begin
repeat
    write ('Introduceţi dimensiunea matricii Mat: n=');
    readln(n);
  until ((n < 11) and (n > 0));
  for i:=1 to n do
  begin
    write ('Introduceţi elementul Mat [', i, ']=');
    readln (Mat[i]);
  end;
```

```
if ((n mod 2)=0) then
   begin
      for i:=1 to (n div 2) do
         begin
               buf:=Mat[i];
               Mat [i]:=Mat [n+1—i];
               Mat [n+1—i]: = buf;
         end
   end
else
      for i:=1 to ((n—1) div 2) do
         begin
               buf:=Mat [i];
               Mat [i]:=Mat [n+1—i]
               Mat [n+1—i]:=buf:
         end;
  for i:=1 to n do
      writeln ('Mat [', i, ']=', Mat [i]);
   readln;
end.
```

7.

```
program corectare;
type vector=array [1 . .4] of real ;
var a:vector;
     max:real;
     i, n:integer;
begin
repeat
     write ('Introduceţi dimensiunea vectorului a: n=');
     readln (n);
until ((n < 5) and (n > 0));
for i:=1 to n do
     begin
          write ('a[', i, ']=');
          readln (a[i]);,
     end;
max:=a[1];
for i:=1 to n—1 do
     if a[i] < a[i+1] then
          max:=a[i+1];
writeln ('Maximul este egal cu: max=', max:3:2);
readln;
end.
```

Greşelile făcute sunt:
— declaraţia de tip a vectorului se face cu '=' şi nu cu ':';
— dimensiunea vectorului trebuie definită;
— operaţiile de atribuire se fac cu ':=' şi nu '='.
— variabila n nu este definită;
— la primul ciclu **for** lipsesc cuvintele cheie **begin** şi respectiv **end** pentru delimitarea corpului ciclului;
— lipseşte un apostrof la instrucţiunea write;

— un element al unei matrici a se apelează cu a[i] şi nu cu a(i);
— variabila max trebuie comparată cu a[i] şi nu cu a(1) acelaşi lucru fiind greşit atât din punct de vedere logic cât şi sintactic.

10.

```
program temperatură;
type matrice=array [1..50, 1..50] of real;
     vector=array [1..31] of real;
     vector 1=array [1..31] of integer;
var a:matrice;
     medie, poz:vector;
     arhiva: vector 1;
     i, j, minzi, minora, buf1, maxzi, maxora:integer;
     min, max, sum, buf: real;
     flag:boolean;
begin
for i:=1 to 24 do
   for j:=1 to 31 do
      begin
         write ('a[', i, ', ', j, ']=')
         readln (a[i, j]);
      end;
max:=a[1, 1];
min:=a[1, 1];
for i:=1 to 24 do
   for j:=1 to 31 do
      begin
         if a[i, j] > max then
            begin
               max:=a[i, j];
               maxora:=i;
               maxzi:=j
            end;
         if a[i, j] < min then
            begin
               min:=a[i, j];
               minora:=i;
               minzi:=j;
            end;
      end;
if a[1, 1]=min then
   begin
               minora:=1;
               minzi:=1;
   end;
if a[24, 31]=max then
   begin
               maxzi:=31;
               maxora:=24;
   end;
if a[1, 1]=max then
```

```
        begin
                maxzi:=1;
                maxora:=1;
        end;
if a[24, 31]=min then
        begin
                minzi:=31;
                minora:=24;
        end;
writeln ('Temperatura maximă a lunii a fost:', max:3:1, 'C în ziua:', maxzi,' la ora:',
maxora);
writeln ('Temperatura minimă a lunii a fost:', min:3:1, 'C în ziua:', minzi, 'la ora:',
minora);
for i:=1 to 31 do
        medie [i]:=0;
for i:=1 to 31 do
        begin
           sum:=0;
           for j:=1 to 24 do
                   sum:=sum+a[i, j];
           medie [i]:=sum/2;
        end
for i:=1 to 31 do
        arhiva [i]:=i;
flag:=false;
while (flag=false) do
        begin
           flag:=true;
           for i:=1 to 31 do
              if medie [i] < medie[i+1] then
                 begin
                      buf:=medie[i];
                      buf1:=arhiva [i];
                      medie [i]:=medie [i+1];
                      arhiva [i]:=arhiva [i+1];
                      medie [i+1]:=buf;
                      arhiva [i+1]:=buf1;
                      flag:=false;
                 end;
        end;
writeln ('Temperatura medie în zilele lunii august a fost:');
for i:=1 to 31 do
      begin
           write ('Ziua:', arhiva [i]:2, 'Temp medie:', medie [i]:3:2);
           if (i mod 3)=0 then
                   writeln;
      end;
readln;
end.
```

14.

```
program matrici;
type matrice=array [1..4, 1..4] of real;
```

```
var a:matrice;
    i, j, n, flag, flag1:integer;
begin
flag:=—1;
repeat
      write ('Introduceţi dimensiunea matricei a : n=');
      readln (n);
until ((n < 5) and (n > 0));
for i:=1 to n do
   for j:=1 to n do
      begin
         write ('a[', i, ', ', j, ']=');
         readln (a[i, j]);
      end;
for i:=i to n— 1 do
   for j:=i+1 to n do
      if (a[i, j]=a[j, i]) then
               flag:=0;
flag1:=0;
for i:=2 to n do
   for j:=1 to i—1 do
      if a[i, j] <> 0 then
               flag1:=1;
   if flag1=0 then
               flag:=1;
flag1:=0;
for i:=2 to n do
   for j:=1 to i—1 do
      if a[j, i] <> 0 then
               flag1:=1;
      if flag1=0 then
               flag:=2;
flag1:=0;
for i:=1 to n do
   for j:=1 to n do
      if a[i, j] <> 0 then
               flag1:=1;
      if flag1=0 then
               flag:=3;
case flag of
   0:writeln ('Matrice simetrică faţă de diagonala principală');
   1:writeln ('Matrice superior triunghiulară');
   2:writeln ('Matrice inferior triunghiulară');
   3:writeln ('Matricea nulă');
else
      writeln ('Matrice oarecare');
end;
readln;
end.
```

16.

```
program căutare;
var succes:boolean;
    n, v, i:integer;
    x     :array[1..1000] of integer;
begin
  write ('Introduceţi dimensiunea şirului X:');
  readln(n);
  write ('Introduceţi valorile componentelor şirului X:');
      for i:=1 to n do
          read(x[i]);
      write ('Introduceţi valoarea V:');
      readln (v);
      succes:=false;
      for i=1 to n do
        if x[i]=v then
          begin
            succes:=true;
            writeln ('Valoarea', v, ' se află în poziţia', i)
          end;
        if not succes then
        writeln ('Valoarea', v, 'nu există în şir')
end.
```

19.

```
program normare;
type vector=array [1..4] of real;
var a:vector;
    max:real;
    i, n:integer;
    begin
    repeat
        write ('Introduceţi dimensiunea vectorului a : n=');
        readln(n);
    until ((n < 5) and (n > 0));
    for i:=1 to n do
      begin
        write ('a[', i, ']=');
        readln (a[i]);
      end;
    max:=a[1];
    for i:=1 to n—1 do
      if a[i] < a[i+1] then
          max:=a[i+1];
    for i:=1 to n do
          a[i]:=a[i]/max;
  for i:=1 to n do
        write ('a[', i, ']=', a[i]:3:2);
        readln;
end.
```

5. Varianta corectă este:

```
function F3 (var a:real):integer;
  begin
    F3:=a+5
  end;
```

6. b) 35 și 'E' reprezintă constante de tip real, respectiv char și deci nu pot fi utilizate ca parametri variabilă (pe care, de exemplu, procedura ar putea dori să-i modifice).
   c), d) Numărul de parametri actuali este mai mic decât numărul de parametri formali.
   e) 3.0 este o valoare de tip real în timp ce parametrul formal corespunzător este de tip integer.

7.

```
var A, B:real;
function Putere (x:real):real;
  var i, p:integer;
  begin
    p:=1;
    for i:=1 to 4 do
    p:=p*x;
    Putere:=p
  end;
...
writeln (A:8:2, '+', B:8:2,'=', Putere (A+B):10:2);
...
```

8.

```
function cmmdc (a, b:integer):integer;
  var t:integer;
  begin
    while b<> 0 do
      begin
        t:=b,
        b:=a mod b;
        a:=t
      end;
    cmmdc:=a
  end;
```

9.

```
type Tablou=array [1..10] of integer;
...
function Minim (A: Tablou; n: integer):integer;
  var i, Min:integer;
  begin
    Min:=A[1];
    for i:=2 to n do
      if Min > A[i] then Min:=A[i];
    Minim:=Min
  end;
```

10. Considerând că a fost deja scrisă funcția cmmdc ce calculează cel mai mare divizor comun pentru două numere întregi precizate (vezi exemplul 12), se va defini următoarea funcție:

```
function cmmmc (a, b:integer):integer;
  begin
    cmmmc:=a*b div cmmdc (a, b)
  end;
```

11.

```
function SumaCifre (n:integer):integer;
  var suma:integer;
  begin
    suma:=0;
    while n div 10 <> 0 do
      begin
        suma:=suma+n mod 10;
        n:=n div 10
      end;
    SumaCifre:=suma+n
  end;
```

12.

```
type Tablou=array [1...10] of integer;
...
function SumaTablou (A:Tablou; n:integer):integer;
  var i, suma: integer;
  begin
    suma:=0;
    for i:=1 to n do
      suma:=suma+A[i];
    SumaTablou:=suma
  end;
```

17. Procedura se va scrie asemănător soluției propuse pentru problema 11) din cadrul capitolului 9.

Capitolul 11

1.

```
program Hermite;
var
        ordin: integer;
        parametru:real;
        h1, h2: real;
function Hermite1 (x:real; n:integer): real;
{Varianta iterativă}
var
        hn, hn1, hn2:real;
        i:integer;
begin
        if (n=0) then
            Hermite1:=1
        else
            if (n=1) then
                Hermite1:=2*x
```

```
        else
            begin
                hn1:=1;
                hn:=2*x;
                for i:=2 to n do
                  begin
                      hn2:=hn1;
                      hn1:=hn;
                      hn:=2*x*hn1—2*(i—1)*hn2
                  end;
                Hermite1:=hn
            end;
end;
function Hermite2 (x:real; n:integer):real;
{Varianta recursivă}
begin
    if (n=0) then
        Hermite2:=1
    else
        if (n=1) then
            Hermite2:=2*x
        else
            begin
                Hermite2:=2*x*Hermite2(x, n—1)—2*(n—1)*Hermite2(x, n—2)
            end;
end;
begin
    write („Precizaţi ordinul polinomului:');
    readln (ordin);
    write ('Precizaţi valoarea lui x:');
    readln (parametru);
    h1:=Hermite1 (parametru, ordin);
    h2:=Hermite2 (parametru, ordin);
    writeln ('Iterativ:', h1:7:3);
    writeln ('Recursiv:', h2:7:3)
end.
```

2.

```
program SumaRec;
var n:integer;
function ParImpar (i, tip:integer):real;
{tip=0 pt. par; tip=1 pt. impar}
begin
  if i=1 then
      ParImpar:=2—tip
  else
      ParImpar:=ParImpar (i—1, tip)*(2*i—tip);
end;
function Suma (n:integer):real;
  begin
    if n=1 then Suma:=0.5
    else
        Suma:=ParImpar (n, 1)/ParImpar (n, 0)+Suma (n—1)
  end
```

```
begin
  readln(n);
  writeln ('Suma este', Suma (n):7:2);
  readln
end.
```

3.

```
program Combinări;
var
    n, p, i:integer;
function Comb (n, k:integer):integer;
  begin
    if k=0 then Comb:=1
    else Comb:=(n—k+1)*Comb (n, k—1) div k
  end;
begin
  write ('Precizați n:');
  readln(n);
  write ('Precizați k:');
  readln(k);
  for i:=1 to p do
  writeln ('Combinari de', n:4, 'luate câte', i:4, '=', Comb(n, i):6);
  readln
end.
```

5.

```
program Regine;
var
R:array [1..8] of 1..8;
Lin: array [1..8] of boolean;
Diag1, Diag2: array [1..15] of boolean;
i:integer;
Suc:boolean;
procedure Încerc (k:integer);
var L:integer;
begin
L:=0;
Suc:false;
repeat
  L:=L+1;
  if lin[L] and Diag1[L+k—1] and Diag2[L—k+8] then
    begin
      r[k]:=L;
      lin[L]:=false;
      Diag1[L+k—1]:=false;
      Diag2[L—k+8]:=false;
      if k < 8 then
        begin
          Încerc (k+1);
          if not suc then
            begin
              lin[L]:=true;
              Diag1[L+k—1]:=true;
```

```
                    Diag2[L—k+8]:=true
                  end;
               end;
            else
               suc:=true
         end;
      until suc or (L=8)
   end;
   begin
      for i:1 to 8 do lin[i]:=true;
      for i:=1 to 15 do
         begin
            Diag1[i]:=true;
            Diag2[i]:=true;
         end;
      Încerc(1);
      if suc then
         begin
            writeln;
            write ('O soluţie este:');
            for i:=1 to 8 do
               write (R[i]:2);
            writeln
         end;
   end.
```

Capitolul 12

5. Varianta corectă este dată în continuare. Comentariile precizează erorile comise în textul iniţial.

```
type
   complex=record
               re:real; {s-a declarat identificator de câmp cu nume rezervat}
               imaginar:real; {s-a omis separatorul „;"}
               end;
var
   c1, c2, c3:complex; {s-a omis variabila c3 ce apare în programul principal}
begin
{  nu se pot citi cu ajutorul procedurii readln alt tip de date decât cele predefinite
   de limbajul Pascal}
   write ('C1. Real=');
   readln (c1. re);
   write ('C1. Imaginar=');
   readln (c1. imaginar);
   write ('C2. Real=');
   readln (c2. re);
   write ('C2. Imaginar=');
   readln (c2. imaginar);
   c3. re:=c1. re+c2. re;
   c3. imaginar:=c1. imaginar+c2. imaginar;
{
— în expresiile decizionale if. .then, pe poziţia condiţiei de testat s-a folosit opera-
   torul de atribuire
```

— s-a introdus separatorul „;" după instrucţiunea ce urmează după „then"
}

```
    if c3.re=0 then
        if c3.imaginar=0 then
            writeln ('Numărul este 0')
        else
            writeln ('Numărul este imaginar')
    else
        if c3. imaginar=0 then
            writeln ('Numărul este real')
        else
            writeln (c3. re, '+i', c3. imaginar);
end.
```

6.

Varianta corectă este:

```
    program Corect ;
type
    rec=record
        x:integer;
        y:real;
      end;
var
    r:rec;
    f:file of rec;
    i:integer;
begin
    assign (f, 'test. pas');
    reset (f);
    read (f, r);
    writeln (r.x:5);
    read (f, r);
    writeln (r.y:5:2);
    close (f);
end.
```

9.

```
program Grupa;
const
    nrstud=30;
    nrmat=8;
type
    catalog=record
            nume:string [20];
            prenume:string[10];
            note:array [1 . .nrmat] of 0 . .10;
            media:real;
            afis;boolean;
        end;
var
    stud1:catalog;
```

```
        cat_fiş:file of catalog;
        c, medie, contor, n, m, i, j, j_1, l1;integer;
        max, max1:real;
        studn, studn_1:string[20];
        studp, studp_1: string [10];
begin
        assign (cat_fiş, 'catalog. dat');
        write ('Introduceţi nr. studenţi:');
        readln (n);
        write ('Introduceţi nr. materii:');
        readln (m);
        writeln;
        rewrite (cat_fiş);
        for i:=i to n do
            begin
                write ('Nume:');
                readln (stud1.nume);
                write ('Prenume:');
                readln (stud1. prenume);.
                c:=0;
                medie:=0;
                for j:=1 to m do
                    begin
                      if j=1 then
                           write ('Prima notă:')
                      else
                           write ('A', j, '—a nota:');
                      readln (stud1.note [j]);
                      if stud1.note [j]=0 then
                           c:=c+1
                      medie:=medie+stud1.note [j];
                    end;
                    if c<=1 then
                      stud1.media:=medie/(m—c)
                    else
                      stud1.media:=0;
                stud1.afiş:=false;
                write (cat_fiş, stud1);
                writeln;
            end;
        close (cat_fiş);
        writeln ('AFIŞARE ÎN ORDINEA MEDIILOR');
        writeln;
        reset (cat_fiş);
        i:=1;
        while i<=n do
            begin
                max:=—1;
                max1:=—1;
                contor:=0;
                while not eof (cat_fiş) do
                    begin
```

```
                    read (cat_fiș, stud1);
                    contor:=contor+1;
                    if (stud1.media>max) and (stud1.afiș=false) then
                        begin
                            max:=stud1.media;
                            max1:=max;
                            studn:=stud1.nume;
                            studp:=stud1.prenume;
                            j:=contor;
                        end;
                    if (stud1.media=max1) and (stud1.afiș=false) then
                        begin
                            studn_1:=stud1.nume;
                            studp_1:=stud1.prenume;
                            j_1:=contor;
                        end;
                    if (studn_1 <studn) or ((studn_1=studn) and
                        (studp_1 <studp)) then
                        begin
                            studn:=studn_1;
                            studp:=studp_1;
                            j:=j_1;
                        end;
                end;
            seek (cat_fiș, j—1);
            if max>0 then
                begin
                    l1:=length (studn)+length (studp);
                    writeln (studn, ' ', studp, ' ', max:32—l1:2);
                    stud1.afiș=true;
                    stud1.nume:=studn;
                    stud1.prenume:=studp;
                    write (cat_fiș, stud1);
                    seek (cat_fiș, 0);
                    i:=i+1;
                end;
            else
                    i:=n+1;
        end;
    readln;
    close (cat_fiș);
end.
```

10.

```
program    Statistica;
{
  se presupune că datele referitoare la fiecare individ sunt inscrise intr-un fișier cu
  tipul „individ" iar citirea lor se face cu ajutorul procedurii „read_data"
}
const
  nr_max        =200; {numărul maxim de persoane eșantionate}
```

```
    necăsătorit   =1;    {codificarea statutului social}
    căsătorit     =2;
    divorţat      =3;
    văduv         =4;
type
    individ=record
              vârsta:integer;
              înălţime:integer;
              statut:integer;
              sex:boolean; {M=true/F=false}
            end;
    eşantion=array [1..nr_max] of individ;
    data_file=file of individ;
procedure read_data (from:string; var nr:integer; var buff:eşantion);
{
    Procedura read_data realizează citirea datelor dintr-un fişier de pe disc.
    Date de intrare:  — numele fişierului → from:string;
    Date de ieşire:   — numărul de indivizi din lot → nr:integer;
                      — vectorul cu datele citite din fişier → buff:eşantion;
    În cazul în care fişierul specificat nu este valid se va realiza ieşirea din program cu
    ajutorul instrucţiunii „halt". De asemenea se va ieşi din program dacă vectorul
    este vid.
}
var
    f:data_file;
    c:char;
    date:individ;
    i:integer;
begin
    assign (f, from);
    reset (f);
    if IOResult <> 0 then
        begin
          writeln ('Eroare la deschiderea fişierului', from);
          write ('Apăsaţi o tastă pentru a opri programul:');
          c:=readkey;
          halt;
        end;
    i:=1;
while not (eof (f)) do
        begin
          read (f, date);
          buff[i]:=date;
          inc (i);
        end;
    nr:=i—1;
    close (f);
    if nr=0 then
        begin
            writeln ('Vectorul de date este vid. Program oprit');
            halt;
        end;
```

```
end; {read_data}
function căsătoriţi (n:integer; e:eşantion):real;
{
   Funcţia întoarce procentul de persoane căsătorite din eşantionul considerat.
   Parametri intrare :  n — numărul de persoane din eşantion
                        e — vectorul cu date persoane
   Întoarce un număr real în intervalul [0 1].
}
var
   i:integer;
   nc:integer;
begin
   nc:=0;
   for i:=1 to n do
        if e[i]. statut=căsătorit then
        inc (nc);
   căsătoriţi:=nc/n;
end; {căsătoriţi}
function sdv (n:integer; e:eşantion; lim_inf, lim_sup: integer; var nsel:integer;
                var sel:eşantion):real;
{
   Funcţia : Selectează După Vârstă  (s.d.v.)
   Funcţia întoarce procentajul de persoane ce au vârsta cuprinsă în intervalul (lim_inf
   . .lim_sup) — deschis la ambele capete.
   Parametri intrare : n — numărul de persoane din eşantion
                       e — vectorul cu date indivizi
   Parametri ieşire: nsel — numărul de persoane găsite cu condiţia îndeplinită
                     sel  — vectorul persoane cu condiţia îndeplinită
   Întoarce un număr real în intervalul [0, 1].
   Observaţie :— Funcţia contorizează acele înregistrări e[i] care îndeplinesc condiţia
                lim_inf < e[i]. vârsta < lim_sup.
                — În cazul în care se doreşte selectarea acelor e[i] pentru care e[i].
                vârsta > lim_inf  (sau < lim_sup),  se  fixează  lim_sup  la  valoarea
                255 (vârsta deocamdată intangibilă...) respectiv lim_inf la valoarea 0.
                — Dacă se doreşte o inegalitate de forma:
                lim_inf<=e[i]. vârsta <=lim_sup, se comunică funcţiei ca parametri
                de apel, lim inf—1 respectiv lim_sup+1.
}
var
   nv:integer;
   i:integer;
begin
   nv:=0;
   for i:=1 to nr_max do
        with sel [i] do
          begin
             vârsta:=0
             înălţime:=0;
             statut:=0;
          end;
   for i:=1 to n do
        if (e[i].vârsta > lim_inf) and (e[i].vârsta < lim_sup) then
```

```
            begin
              inc(nv);
              sel[nv]:=e[i];
            end;
          nsel:=nv;
          sdv:=nv/n;
      end; {sdv}
{VARIABILELE PROGRAMULUI PRINCIPAL}
var
      es, esm:eşantion;
      nr, nrm:integer;
      i, nr_bărbaţi:integer;
      suma, m:real;
      nume-fiş:string;
      p:real;
{PROGRAM PRINCIPAL}
begin
      write ('Introduceţi numele fişierului ce conţine data:');
      readln (nume_fiş);
      read_data (nume_fiş, nr, es);
      p:=căsătoriţi (nr, es)*100;
      writeln ('Din lotul considerat', p:4:2, '% sunt căsătoriţi');
      p:=sdv (nr, es, 0, 18, nrm, esm)*100; {persoane cu vârsta < 18 ani}
      writeln (p:4:2, '% au vârsta sub 18 ani');
      p:=sdv (nr, es, 17, 63, nrm, esm)*100; {18 < =vârsta <=62 ani}
      writeln (p:4:2, '% au vârsta cuprinsă între 18 şi 62 ani');
      p:=sdv (nr, es, 62, 255, nrm, esm)*100;
      writeln (p:4:2, '% au vârsta peste 62 ani');
      p:=sdv (nr, es, 17, 31, nrm, esm)*100; {vârste între 18 şi 30 ani}.
{se vor lua în considerare numai cei de sex masculin la calculul înălţimii medii}
  suma:=0;
  nr_bărbaţi:=0;
  for i:=1 to nrm do
        if esm [i].sex then
        begin
           suma:=suma+esm[i].înălţime;
           inc (nr_bărbaţi);
        end;
  if nr_bărbaţi=0 then
        begin
           writeln ('Nu există date de prelucrat. Program oprit');
           writeln ('Apăsaţi tasta Enter'); readln;
           halt;
        end;
     m:=suma/nr_bărbaţi;
     writeln (m:3:2, ' cm înălţimea med. bărbaţi între 18 şi 30 ani');
  readln;
end.
```

14.

o variantă de rezolvare este următoarea :

```
type
  complex=record
            re:real;
            im:real;
          end;
procedure addc (var x:complex; y:complex); {x=x+y}
begin
  x.re:=x.re+y.re;
  x.im:=x.im+y.im;
end;
procedure mulc (var x:complex; y:complex); {x=x*y}
var
  buff: complex;
begin
  buff:=x;
  x.re:=buff.re*y.re—buff.im*y.im;
  x.im:=buff.re*y.im+buff.im*y.re;
end;
procedure divc (var x:conplex; y:complex) {x=x/y}
var
  buff:complex;
  modul:real;
begin
  buff:=x;
  modul:=sqr(y. re)+sqr(y. im);
  if modul=0.0 then
    begin
      writeln ('Împărţire cu 0!');
    halt;
    en'
  x. re:=1/modul*(buff. re*y. re+buff. im*y. im);
  x. im:=1/modul*(buff. im*y. re+buff. re*y. im);
end;
procedure subc (var x:complex; y:complex); {x=x—y}
begin
  x. re:=x. re—y. re;
  x. im:=x. im—y. im;
end;
function module (x complex):real; {module=sqr(re)+sqr(im)}
begin
  module:=sqr(x. re)+sqr(x. im);
end;
procedure conj(x:complex; var conjx:complex);
                              {conjx=x. re—i*x. im}
begin
  conjx. re:=  x. re;
  conjx. im:=— x.im;
end;
```

```
function re (x:complex) real; {intoarce partea reală}
begin
  re:=x.re;
end;
function im(x:complex):real; {intoarce partea imaginară}
begin
  im:=x.im;
end;
function arg (x.complex):real;
                          {intoarce argumentul numărului complex}
begin
  if x.re=0 then arg:=pi/2 else arg:=arctan (x.im/x.re);
end;
procedure writec (x:complex); {afişează un număr complex}
begin
  write (x.re:4:3);
  if x.im > 0 then writeln('+i*', x.im:4:3) else
  if x.im < 0 then writeln ('—i*', abs (x.im):4:3);
end;
var
  x, y, z:complex;
 begin
  {exemple de utilizare}
  x.re:=2;
  x.im:=0;
  y.re:=3;
  y.im:=—2;
  writec (x);
  writec(y);
  conj (x, z);
  writec (z);
  addc (x, y);
  writec (x);
  mulc (y, x);
  writec (y);
  divc (x, y);
  writec (x);
  writeln (arg(x):4:3);
  readln;
end.
```

15.

```
program Concatenare;
{La scrierea acestui program s-au folosit proceduri şi funcţii din biblioteca Turbo
Pascal}
type
  fişier=file of real;
  num_fiş=string [12];
var
  f1, f2, f3, fr:fişier;
   nr, nr1:real;
  fiş1, fis2, fis3, fis_rez:num_fiş;
begin
      write ('Nume fişier 1=');
```

```
        readln (fiş1);
        write ('Nume fişier 2=');
        readln (fiş2);
        write ('Nume fişier 3=');
        readln (fiş3);
        write ('Nume fişier rezultat după concatenare');
        readln (fiş_rez);
        assign (f1, fiş1);
        reset (f1);
        assign (f2, fiş2);
        reset (f2);
        seek (f1, filesize (f1));
        while not eof(f2) do
           begin
                 read (f2, nr);
                 write (f1, nr);
           end;
        close (f2);
        assign (f3, fiş3);
        reset (f3);
        while not eof(f3) do
           begin
                 read (f3, nr1);
                 write (f1, nr1);
           end;
        close (f1);
        assign (fr, fiş_rez);
        reset (fr);
        if (IOResul =0) then
           begin
                 close (fr);
                 erase (fr);
           end;
           rename (f1, fiş_rez);
        close (f3);
end.
```

Capitolul 13

1.

```
function suma (l:list):integer;
var
   s:integer;
begin
   s:=0;
   while l <> nil do
      begin
         if l↑.k > 0 then s:=s+l↑.k;
         l:=l↑.next
      end;
   suma:=s
end;
```

2.

```
procedure tipar (l:list);
var
    par, impar, poz, neg:integer;
begin
    par:=0;
    impar:=0;
    poz:=0;
    neg:=0;
    while l <> nil do
        begin
            if l↑.k mod 2=0
                then par:=par+1
            else
                impar:=impar+1;
            if l↑.k > 0 then
                poz:=poz+1
            else
                neg:=neg+1;
            l:=l↑.next
        end;
    writeln ('În listă sunt', par:2, ' elemente pare',
    impar:2, ' elemente impare,');
    writeln (poz:2, ' elemente pozitive şi ', neg:2 ' elemente negative,')
end;
```

3.

```
function impare (l:list):integer;
var
  k:integer; p:list;
begin
    k:=0;
    if l <> nil then
        begin
            p:=l;
            repeat
            if p↑.k mod 2 <> 0 then k:=k+1;
                p:=p↑.next
            until p=l
        end;
impare:=k
end;
```

4.

Varianta recursivă:

```
function comp (l1, l2:list):boolean;
begin
    if l1=nil then
        comp:=l2=nil
    else
        if l2=nil then
            comp:=false
```

```
        else
            comp:=(l1↑.k=l2↑.k) and comp (l1↑.next, l2↑.next)
    end;
```

Variantă nerecursivă:

```
function comp(l1, l2:list):boolean;
begin
    while (l1 <> nil) and (l2 <> nil) and (l1↑.k=l2↑.k) do
      begin
          l1:=l1↑.next;
          l2:=l2↑.next
      end;
    comp:=(l1=nil) and (l2=nil)
end;
```

5.

```
function member (l:list; k:integer):boolean;
begin
    while (l <> nil) and (l↑.k) <> k) do
        l:=l↑.next;
    member:=l <> nil
end;
```

6. Se va utiliza funcţia „member" de la problema 5.
Intersecţia:

```
function intersecţie (a, b:list):list;
var
    c:list,
begin
    c:=nil;
    while a <> nil do
        begin
            if member (b, a↑.k) then c:=cons (a↑.k, c);
            a:=a↑.next
        end;
    intersecţie:=c
end;
```

Diferenţa: diferă de intersecţie într-un singur loc, anume în condiţia instrucţiunii if, care devine acum:

```
if not member (b, a↑.k) then c:=cons (a↑.k, c);
```

Reuniunea;

```
function reuniune (a, b:list):list;
var
    c:list;
begin
{se introduc în c toate elementele din a, necondiţionat:}
    c:=nil;
  while a <> nil do
        begin
            c:cons (a↑.k, c);
```

```
                a:=a↑.next
            end;
        {se introduc în c şi toate elementele din b—a:}
        while b <> nil do
            begin
                if not member (a, b↑.k) then c:=cons (b↑.k, c);
                b:=b↑.next
            end;
        reuniune:=c;
    end;
```

7.

```
function ins (k:integer; l:list):list;
begin
    if (l=nil) or (k<=l↑.k) then
        ins:=cons (k, l)
    else
        begin
            l↑.next:=ins(k, l↑.next);
            ins:=l
        end;
end;
```

8. Se poate folosi metoda de sortare prin inserţie; soluţia propusă utilizează funcţia „ins" de la problema 7:

```
function sort (l:list):list;
var
    r:list,
begin
    r:=nil;
    {se inserează, rând pe rând, toate cheile lui l în lista r, păstrând însă lista r
    ordonată crescător}
    while l <> nil do
      begin
            r:=ins(l↑.k, r);
            l:=l↑.next
        end;
    sort:=r
end;
```

9. Soluţia recursivă este cea mai naturală, deoarece lista trebuie construită de la coadă la cap:

```
function lista (n:integer):integer;
begin
    if n=0 then
        lista:=nil
    else
        lista:=cons (n mod 10, lista (n div 10))
    end;
```

10. Metoda 1 — iterativă:
Se folosesc: o procedură („conslast") şi 2 funcţii auxiliare („v" şi „n"):

```
procedure conslast (k:integer; var r, rlast:list);
{adaugă cheia „k" la sfârşitul listei „r";
rlast=celula de la sfârşitul lui „r" }
begin
    if r=nil then
        begin
            r:=cons (k, nil);
            rlast:=r;
        end;
    else
        begin
            rlast↑.next:=cons (k, nil);
            rlast:=rlast↑.next;
        end;
end;
function v (l:list):integer;
{intoarce valoarea din prima celulă a listei „l"
sau 0 dacă l=nil}
begin
    if l=nil then v:=0
    else v:=l↑.k
end;
function n (l:list):integer;
{intoarce restul listei „l", fără primul element;
întoarce nil, dacă l=nil }
    begin
    if l=nil then n:=nil
    else n:=l↑.next
end;
function sum (l1, l2:list):list;
var
    r, rlast:list;
    t:integer; {transportul}
    x1, x2, s:integer;
begin
    r:=nil;
    t:=0;
    while (l1 <> nil) or (l2 <> nil) or(t <> 0) do
        begin
            x1:=v(l1);
            x2:=v(l2);
            s:=x1+x2+t;
            if s >=10 then
                begin
                    s:= s—10;
                    t:=1
                end;
        else t:=0;
      conslast (s, r, rlast);
      l1:=n(l1);
      l2:=n(l2)
```

```
        end;
    sum:=r
    end;
```

Metoda 2 — recursivă:
Folosim o funcție auxiliară sum1, recursivă, care are același rol ca și sum, dar primește un parametru suplimentar: transportul.

```
function sum1 (t:integer; l1, l2:list):list;
var x1, x2, s:integer;
begin
if (l1=nil) and (l2=nil) and (t=0)
    then sum1:=nil
    else begin
        x1:=v(l1);          {aceeași funcție v din prima metodă}
        x2:=v(l2);
        s:=x1+x2+t;
        if s > =10
          then begin
            s:=s—10;
            t:=1
          end;
          else t:=0;
        sum1:=cons (s, sum1 (t, n(l1), n(l2))
                              {aceeași funcție n din prima metodă}
        end;
end;
```

11. Problema este similară problemei 10.
12. Se vor folosi funcțiile auxiliare „v" și „n" din problema 10.
În plus, funcția „prodsum" reprezintă esența soluției:

```
function prodsum (l:list; m, tm:integer; r:list; ts:integer):list;
{înmulțește lista l cu numărul m (având deja transportul de înmulțire tm) și adaugă
rezultatul la lista r (având deja transportul de adunare ts)}
begin
    if (l=nil) and (tm=0) and (ts=0) then
          prodsum:=r
    else
          begin
              if r=nil then r:=cons(0, nil);
              p:=v(l)*m+tm;
              s:=r↑.k+(p mod 10)+ts;
              if s > =10 then
                  begin
                       s:=s—10;
                       ts:=1
                  end;
              else
                  ts:=0;
              r↑.k:=s;
              r.↑next:=prodsum (n(l), m, (p div 10), r↑.next, ts);
              prodsum:=r
```

```
            end;
    end;
    function prod (l1, l2):list;
    begin
        if l2=nil then prod:=nil
        else prod:=prodsum (l1, l2↑.k, 0, cons(0 prod(l1, l2↑. next)), 0)
    end;
```

13. Se aplică schema lui Horner:

$P(x)=a_0+x*a_1+x^2*a_2+...+x^n*a_n=a_0+x*(a_1+x*(a_2+...+x*a_n)...));$

Atunci când anumite puteri lipsesc, formula devine:

$P(x)=a_{g1}*x^{g1}+a_{g2}*x^{g2}+a_{g3}*x^{g3}+...=$

$x^{g1}*(a_{g1}+x^{g2-g1}*(a_{g2}+x^{g3-g2}*(a_{g3}+x^{g4-g3}*(...))))$

unde $g1 < g2 < g3 <...$ sunt gradele corespunzătoare coeficienților nenuli ai polinomului P(x).

```
function pow (x:real: k:integer):real;
{calculează x la puterea k (x**k) efectuând un număr minim de înmulțiri, pe baza
observației :
dacă k=2*k', atunci se calculează y:=x**k', și x**k:=y*y}
begin
    if k=0 then
        pow:=1
    else
        if odd (k) {testează dacă k este impar — odd este funcție standard în
                   Pascal}
            then pow:=x*pow (x, k—1)
        else
            begin
                y:=pow(x, k, div 2);
                pow:=y*y
            end;
end;

function eval1 (p:pol; x:real; g:integer):real;
begin
    if p=nil then eval1:=0
    else eval1:=pow (x, p↑.g—g)*(p↑.c+eval1(p↑.next, x, p↑.g))
end;

function eval (p:pol; x:real):real;
begin
eval:=eval1 (p, x, 0)
end;
```

14 și **15** sunt asemănătoare problemelor **10** și **12**.

Capitolul 17

2. int num=2
3. a) 1,500,000
   b) 0.000,001,5
   c) 7,654.3
   d) −0.007,654,3
4. Operatori + şi %
5. număr++
6. usa += calif
9. a) fals
   b) adevărat (se compară codurile ASCII)
   c) fals
   d) adevărat

11.

0008
0001
0001

12.

```
/*calculează virsta în zile şi ore*/
#include <stdio. h>
main ()
{
        int ani, zile, ore;
        printf ("Introduceţi virsta in ani:");
        scanf ("%d", &ani);
        zile=365*ani;
        ore=24*zile;
        printf ("Aveţi virsta de %d zile.\n", zile);
        printf ("Aveţi virsta de %d ore.\n", ore);
}
```

13.

```
/*unghi.c*/
/*calculează unghiul în grade, minute şi secunde*/
#include <stdio. h>
main ()
{
        fioat rad, var;
        int grad, min, sec;
        printf ("Introduceţi unghiul în radiani:");
        scanf ("%f", &rad);
        var=180*rad/3.14159;
        grad=var;
        var=(var—grad)*60;
        min=var;
        var=(var—min)*60;
        sec=var;
        printf ("Unghiul are %d grade, %d minute şi %d secunde.\n", grad, min, sec);
}
```

Observaţie: dacă **var** este variabila reală şi **grad** este variabila întreagă, prin instrucţiunea de atribuire **grad=var;** se extrage din **var** partea întreagă.

14.

```
/*cub.c*/
/*calculează suma cuburilor a două numere reale*/
#include <stdio.h>
main ()
{
    float x, y, p, q.
    printf ("Introduceţi numerele:");
    scanf ("%f %f", &x, &y):
    p=x*x*x:
    q=y*y*y;
    printf ("Rezultatul este: %f\n", p+q):
}
```

15.

Observaţie: presupunem că şirul care va fi afişat este cuvântul SUCCES; el are 6 caractere.

```
/*centrare.c*/
/*afişează un text centrat*/
#include <stdio. h>
main ()
{
    int n, var;
    printf ("Introduceţi numărul de caractere ale şirului:");
    scanf ("%d", &n):
    var=(80—n)/2;
    var+=n;
    printf ("%*s\n", var, "SUCCES");
}
```

Observaţie: în limbajul C, dimensiunea câmpului de reprezentare a unei date sau a unui şir, precum şi precizia de reprezentare, pot fi specificate ca parametri în funcţia **printf()** şi trataţi ca atare, cu ajutorul caracterului (*) Deci, ele pot apare ca variabile în loc de constante.
De exemplu, în instrucţiunea:

printf(„%*.*f",10,4,123.3);

primul (*) se asociază cu 10 iar al doilea (*) se asociază cu 4. Instrucţiunea este echivalentă cu:

printf("%10.4f",123.3):

Această particularitate am aplicat-o în exemplul nostru.

16.

```
/*caracter.c*/
/*afişează un caracter, adresa lui şi codul lui ASCII*/
#include <stdio. h>
#include <conio. h>
main ()
{
    char ch;
    printf ("Introduceţi cracterul:");
```

```
        ch=getche ();
        printf ("\nCaracterul %c are codul ASCII %3d şi se află la adresa %x.\n", ch,
        ch, &ch);
}
```

Capitolul 18

1.

```
/*sum. c*/
/*calculează suma pătratelor numerelor naturale cuprinse între două numere*/
#include <stdio. h>
main ()
{
        unsigned long int sum=0; /*rezultatul*/
        int x, y; /*limitele*/
        int i;
        printf ("Introduceţi limitele x, y:");
        scanf ("%d %d", &x, &y);
        for (i=x; i< =y; i++)
                sum +=i*i;
        printf ("Rezultatul este %ld\n", sum);
}
```

2.

```
/*numcar. c*/
/*determină numărul caracterelor dintre două caractere date*/
#include <stdio. h>
main ()
{
        char a, b; /*limitele*/
        int d;
        while (1)
                {
                printf ("Introduceţi caracterele a, b: ");
                scanf (" %c %c", &a, &b);
                d=b—a;
                printf ("\n %d\n", d);
                printf ("Între %c şi %c există %d caractere.\n", a, b, b—a);
                }
}
```

3.

```
/*linie.c*/
/*trasează două linii cu (*)*/
#include <stdio. h>
main ()
{
        int n;                  /*numărul de (*)*/
        int i=0, j;
        printf ("Introduceţi n:");
        scanf ("%d", &n);
        while (i<2)
        {
```

```
                j=0;
                while (j<n)
                {
                        printf ("%c", '*');
                        j++;
                }
                printf ("\n");
                i++;
        }
}
```

4.

```
/*pi.c*/
/*calculează valoarea lui π*/
#include <stdio.h>
main ()
{
        float eps;          /*precizia calculului*/
        float pi=0;         /*rezultatul*/
        float val;
        int n=0, i=—1;
        printf ("Introduceţi eps:");
        scanf ("%f", &eps);
        do
        {
            i=—i;
            val=1./(2*n+1);
            pi+=i*val;
            n++;
        }
        while (4*val > eps);
        printf ("Rezultatul este %.2f\n", 4*pi);
}
```

Observaţie: dacă instrucţiunea **val=1./(2*n+1)**, am fi scris-o **val=1/(2*n+1)** (1 este în acest caz număr întreg şi nu real), rezultatul împărţirii ar fi fost un număr întreg. Ca atare, pentru $n \geq 1$, rezultatul împărţirii ar fi fost tot timpul 0.

5.

```
/*polinom. c*/
/*calculează valoarea unui polinom într-un punct*/
#include <stdio. h>
main ()
{
        float x: /*punctul*/
        float rez : /*valoarea polinomului*/
        do
        {
                printf ("\nIntroduceţi valoarea lui x:");
                scanf ("%f", &x);
                rez=9*x*x*x—2*x*x+120*x—130;
                printf ("Rezultatul este %f\n", rez);
```

```
        printf ("Continuaţi? Tastaţi 'y' pentru da şi orice altceva pentru nu:");
    }
    while (getche ()=='y');
    printf ("\n Programul s-a terminat!\n");
}
```

6. Nu; după cuvântul **case** se pun două puncte şi după instrucţiunile **printf()** trebuie puse instrucţiuni **break.**

7. Nu; variabile, precum **temp,** nu pot fi folosite în expresii **case.**

9. Valoarea expresiei este 0.

10.

```
/*retribut. c*/
/*calculează retribuţia unui salariat*/
#include <stdio. h>
main ()
{
        int grupa;
        float salar;
        while (1)
        {
                printf ("Introduceţi grupa şi salariul:");
                scanf ("%d %f", &grupa, &salar);
                if (salar < 0)
                        printf ("Salariul nu poate fi negativ!\n");
                else
                        switch (grupa)
                        {
                                case 1:
                                        printf ("Retribuţia este %f\n", 150+salar);
                                        break;
                                case 2:
                                        printf ("Retribuţia este %f\n", 250+salar);
                                        break;
                                case 3:
                                        printf ("Retribuţia este %f\n", 350+salar);
                                        break;
                                default:
                                        printf ("Nu aţi introdus grupa corect.\n");
                        }
        }
}
```

11.

```
/*ecuaţie2.c*/
/*calculează rădăcinile unei ecuaţii de gradul 2*/
#include <stdio.h>
include <math. h>
main ()
{
```

```
    float a, b c;              /*coeficienţii ecuaţiei*/
    float rad1, rad2;          /*rădăcinile ecuaţiei*/
    double discr;              /*discriminantul ecuaţiei*/
    printf ("Introduceţi coeficienţii a, b, c:");
    scanf ("%f %f %f", &a, &b, &c).
    discr=b*b—4*a*c;
    if (discr < 0)
            printf ("Ecuaţia are rădăcini complexe.\n");
    else
    {
            if (discr==0)
            {
                    printf ("Ecuaţia are rădăcini reale şi confundate.\n");
                    rad1=(—b)/(2*a);
                    rad2=(—b)/(2*a)·
            }
            else
            }
                    printf ("Ecuaţia are rădăcini reale şi distincte.\n");
                    rad1=(—b+sqrt(discr))/(2*a);
                    rad2=(—b—sqrt(discr))/(2*a);
            }
    printf ("rad1=%.2f\n", rad1);
    printf ("rad2=%.2f\n", rad2);
    }
}
```

12.

```
/*viteza.c*/
/*apreciază regimul de viteză al unui conducător auto*/
#include <stdio. h>
main ()
{
        int viteza;
        while (1)
        {
                printf ("Introduceţi viteza:");
                scanf ("%d", &viteza);
                if (viteza < 120)
                        if (viteza < 80)
                                if (viteza< 60)
                                                printf ("Viteza corectă.\n");
                                else
                                {
                                        printf ("Viteza mare în localitate.\n");
                                        printf ("Viteza corectă în afara localităţii.\n");
                                }
                        else
                                printf ("Viteza mare.\n");
                else
                        printf ("Viteza criminală.\n");
        }
}
```

13.

```
/*maxim1. c*/
/*determină maximul unor numere*/
#include <stdio. h>
main ()
{
        int x1, x2, x3, x4, x5;          /*numerele de comparat*/
        int max;
        printf ("Introduceţi numerele: ");
        scanf ("%d %d %d %d %d", &x1, &x2, &x3, &x4, &x5);
        max=(x1 > x2) ? x1 : x2;
        max = (max > x3) ? max : x3;
        max = (max > x4) ? max : x4;
        max = (max > x5) ? max : x5;
        printf ("Numărul maxim este max = %d\n", max);
}
```

Capitolul 19

1.

```
/*ordonare. c*/
/*ordonează crescător trei numere întregi*/
#include <stdio. h>
void order (int, int, int);
main ()
{
        int x1, x2, x3;          /*numerele introduse*/
        printf ("Introduceţi numerele:");
        scanf ("%d %d %d", &x1, &x2, &x3),
        order (x1, x2, x3);
}
void order (int x, int y, int z)
{
        int a, b, c, w;
        do
        {
                a=y—x ; b=z—x; c=z—y;
                if (a < 0)
                        {w=x; x=y; y=w; continue;}
                if (b < 0)
                        {w=x; x=z; z=w; continue;}
                if (c < 0)
                        {w=y; y=z; z=w; continue;}
        }
        while (a < 0 || b < 0 || c < 0);
        printf ("Numerele ordonate crescător:\n");
        printf ("%d %d %d\n", x, y, z);
}
```

2.

```
/*min_max. c*/
/*găsește cel mai mic și cel mai mare număr din cele n numere date*/
#include <stdio. h>
float max (float, float);
float min (float, float);
main
{
      float x1, x2, x3, x4, x5;      /*numerele care se ordonează*/
      float ymin, ymax;
      printf ("Introduceți cele n numere:\n");
      scanf (" %f %f %f %f %f", &x1, &x2, &x3, &x4, &x5);
      ymin=min(x1, x2);
      ymin=min(ymin, x3);
      ymin=min(ymin, x4);
      ymin=min(ymin, x5);
      ymax=max(x1, x2);
      ymax=max(ymax, x3);
      ymax=max(ymax, x4);
      ymax=max(ymax, x5);
      printf ("ymin=%.4f\nymax=%.4f\n", ymin, ymax);
}
float max (float x, float y)
{
      float ymax;
      ymax=(x > y) ?x:y;
      return (ymax);
}
float min (float x, float y)
{
      float ymin;
      ymin=(x > y) ? y:x;
      return (ymin);
}
```

3.

```
/*sistem. c*/
/*rezolva un sistem de ecuații de ordinul 2*/
#include <stdio. h>
#define DET(x, y, z, t) (x*y—z*t)
main ()
{
      float a11, a12, b1;
      float a21, a22, b2;          /*coeficienții*/
      float x1, x2;                /*soluțiile*/
      float det1;
      printf "(Introduceți coeficienții pentru fiecare ecuație\n:");
      scanf ("%f %f %f", &a11, &a12, &b1);
      scanf ("%f %f %f", &a21, &a22, &b2);
      det1=DET(a11, a22, a12, a21);
      if (det1 !=0)
      {
```

```
        printf („Sistem compatibil determinat.\n");
        x1=DET (b1, a22, b2, a21)/det 1;
        x2=DET (a11, b2, a12, b1)/det 1;
        printf ("x1= %.4f\n", x1);
        printf ("x2= %.4f\n", x2);
    }
    else if (DET (a11, b2, a21, b1)==0)
    {
        printf ("Sistem compatibil nedeterminat.\n");
        printf ("x1= %.4f— %.4f*par\n", b1/a11, a12/a11);
        printf ("x2=par\n");
    }
    else
        printf ("Sistem incompatibil.\n");
}
```

4.

```
/*num_e. c*/
/*calculează numărul e*/
#include <stdio. h>
long int fact (int);
main ()
{
    float eps, e=1;
    int n=1;
    printf ("Introduceţi precizia eps:");
    scanf ("%f", &eps);
    do
    {
        e+=1./fact(n);
        n++;
    }
    while (eps < 1./fact(n));
    printf ("Rezultatul este e= %.4f\n", e);
}
long int fact (int n)
{
    long int fct=1;
    while (n > 1)
        fct *=n— —;
    return (fct);
}
```

Capitolul 20

1. Pentru tipărirea a câte 6 numere pe o linie, se testează explicit, după tipărirea elementului i, dacă indicele i (care este iniţializat cu (0) este divizibil prin 5.

```
void tip_tab_f (float t[ ], int n)
{
    int i;
    putchar ('\n');
    for (i=0; i< n; i++) {
```

```
        printf ("%11.3f", t[i]);
        if ((i%6)==5 || i==n-1)
            putchar ('\n');
        else
            putchar (' ');
    }
}
```

2. Se folosește funcția **scanf** pentru citire.

```
void read_tab (int t[], int n)
{
    int i
    for (i=0; i< n; i++) {
        printf ("Element %d=", i);
        scanf ("%d", &t[i]);
    }
}
```

3. Condiția de execuție a ciclului este: caracterul curent să fie diferit de terminatorul '\0' .

```
void my_puts (char s[ ])
{
    int i;
    for (i=0; s[i] != '\0'; i++)
        putchar (s[i]);
    putchar ('\n');
}
```

4. Funcția **randomize** se apelează o singură dată, apoi se folosește funcția **random.** Trebuie inclus fișierul header **stdlib.h,** care conține prototipurile acestor funcții.

```
void init_tab (int t[ ], int n)
{
    int i;
    randomize ( );
  for (i=0; i< n; i++)
        t[i]=random (10000);
}
```

5. Funcția se scrie transpunând direct algoritmul prezentat.

```
void sort1 (int a[ ], int n)
{
    int sortat, i, temp;
    sortat=0;
    while (!sortat) {
        sortat=1;
        for (i=0; i< n-1; i++)
            if (a[i] > a[i+1]) {
                temp=a[i];
                a[i]=a[i+1];
                a[i+1]=temp;
```

```
                    sortat=0;
                }
        }
}
```

O funcție de tipărire a unui tablou de întregi, după modelul celei de la exercițiul 1, este:

```
void tip_tab_i (int t[ ], int n)
{
        int i;
        putchar ('\n');
        for (i=0; i < n; i++) {
                printf ("%7d", t[i]);
                if ((i%10)==9 || i==n— 1)
                        putchar ('\n');
        else
                putchar (' ');
        }
}
```

Programul principal trebuie să fie de forma:

```
#include <stdio. h>
#include <stdlib. h>
#define NMAX 50
int x[NMAX];
void main (void)
{
        init_tab (x, NMAX);
        sort1 (x, NMAX);
        tip_tab_i(x, NMAX);
}
```

O îmbunătățire a funcției de sortare se poate face în felul următor: în loc de a memora într-o variabilă de tip logic (cu valori 1 sau 0) dacă tabloul este sau nu sortat, este mai eficient să se memoreze, într-o variabilă **marcaj**, indicele ultimului element care a fost schimbat cu vecinul său din dreapta. Variabila **marcaj** este inițializată la fiecare parcurgere a tabloului cu valoarea **—1** (care nu poate fi un indice de tablou). Parcurgerea tabloului se face de la indicele 0 până la un indice numit **ultim**, care este inițializat cu **n—1** și, la fiecare trecere, asignat cu valoarea **marcaj** (indicele ultimului element schimbat). Această tehnică se bazează pe observația că, dacă la o parcurgere nu s-a făcut nici o interschimbare de la un indice **k** până la sfârșitul tabloului, elementele respective sunt deja sortate și nu mai trebuie examinate la parcurgerile următoare. Funcția se încheie dacă într-o parcurgere nu s-a făcut nici o interschimbare (deci **marcaj** este —1). Această variantă de metodă de sortare se numește **metoda bulelor (bubblesort)**, deoarece elementele cu valori mari se deplasează către sfârșitul tabloului, într-un mod asemănător cu ridicarea unor bule de gaz într-un vas cu lichid.

```
void sort2(int a[ ], int n)
{
        int ultim, i, marcaj, temp;
```

```
ultim = n—1;
while (1) {
        marcaj = — 1;
                for (i=0; i < ultim; i++) {
                if (a[i] > a[i+1])  {
                        temp=a[i];
                        a[i]=a[i+1];
                        a[i+1]=temp;
                        marcaj=i;
                }
        }
        if (marcaj == —1)
                break;
        else
                ultim = marcaj;
        }
}
```

7. x=2 y=1
   x=3 y=2

8. x=0 y=0
   x=—1 y=—1

10. Pointerii de tip **(void *)** trebuie convertiţi explicit (prin **cast**) la pointer de tip **(char *)** înainte de a fi dereferenţiaţi (dimensiunea obiectelor este dată în octeţi).

```
#include <stdio. h>
void schimbă (void *a, void *b, size_t n)
{
        char temp;
        while (n— —) {
                temp= * (char *)a);
                *((char *)a)++= *((char *)b);
                *((char *)b)++= temp;
        }
}
void main (void)
{
        int i1=1, i2=2;
        float f1=2.5 f2=4.7;
        printf ("i1= %d i2= %d\n", i1, i2);
        schimbă ( &i1, &i2, sizeof (int));
        printf ("i1= %d i2= %d\n", i1, i2);
        printf ("f1= %f f2= %f\n", f1, f2);
        schimbă ( &f1, &f2, sizeof (float));
        printf ("f1= %f f2= %f\n", f1, f2);
}
```

11. Programul va tipări adresele şi valorile elementelor din tablou.

12. Găsim întîi sfârşitul primului şir, oprindu-ne chiar pe terminatorul '\0', apoi copiem toate caracterele din al doilea şir în primul şir, începând de la poziţia găsită; se copiază inclusiv terminatorul din al doilea şir. Se

presupune că la sfârşitul primului şir există rezervat spaţiu suficient pentru a concatena al doilea şir.

```
void strcat (char *a, char *b)
{
        while (*a)
            a++;
        while (*a++= *b++)
            ;
}
char a[20]="abcde";             /* Se rezervă 20 de octeţi dar se */
                                /* iniţializează numai primii 6 */
char *b="fghijk";
void main (void)
{
      puts(a); puts (b);
      stricat (a, b);
      puts (a); puts (b);
}
```

13. Adresa elementului găsit este **tab+mijloc.**

```
int *caută_bin(int tab[ ], int obiect, int n)
{
      int comp, stinga=0, mijloc, dreapta=n—1;
      while (stinga <=dreapta) }
            mijloc=(stinga+dreapta)/2;
            if ((comp=obiect—tab[mijloc] < 0)
                  dreapta=mijloc—1;
            else
                  if (comp > 0)
                        stinga=mijloc+1;
                  else
                        return tab+mijloc;
      }
      return NULL;
}
```

16. Se face comparaţia şirurilor cu funcţia **strcmp,** care întoarce 0 dacă şirurile coincid.

```
#include <stdio. h>
#include <string. h>
int caută_sir (char *tab[ ], int n, char *obiect)
{
      int i;
      for (i=0; i< n; i++)
            if (strcmp (obiect, tab[i])==0)
                  return i;
            return —1;
}
char *tab_sir[5]={
            "12346790", "abcdef", "qwerty", "ABCDEF", "1993"
};
char *sir1="ABCDEF";
```

```
char *şir2="Acest şir nu va fi găsit";
void main (void)
{
        int i;
        i=caută_şir (tab_şir, 5, şir1);
        printf ("Poziţie= %d\n", i);
        i=caută_şir (tab_şir, 5, şir2);
        printf ("Poziţie= %d\n", i);
        i=caută_şir (tab_şir, 5, "Un şir constant");
        printf ("Poziţie= %d\n", i);
}
```

17. Funcţia **read_lines** întoarce numărul de linii efectiv introduse, testând dacă şirul introdus este vid (funcţia **gets** nu pune '\n' în şir). Se duplică şirul citit (cu **strdup**) şi se copiază în poziţia curentă din tabloul de pointeri. Operaţia se execută într-o buclă din care se iese dacă s-a atins dimensiunea maximă sau s-a introdus '\n' pe o linie goală.

```
int read_lines (char **a, int nmax)
{
        char s[80];
        int gata=0;
        int n=0;
        do {
                gets(s);
                if (strlen(s)==0)
                        gata=1;
                else {
                        *a++=strdup(s);
                        n++;
                }
        } while (n< nmax && !gata);
        return n;
}
```

Funcţia de sortare foloseşte un pointer **p** de tip (char * *), iniţializat la fiecare parcurgere a tabloului cu adresa sa de început. Pointerul **p** este incrementat pe măsură ce se parcurg elementele tabloului. Bucla internă se execută de **n−1** ori, iar două elemente vecine ale tabloului se accesează prin expresiile *****p** şi *****(p+1)**.

```
void sort_lines (char **a, int n)
{
        int gata=0, m;
        char *t;
        char **p;
        while (!gata) {
                gata=1; p=a; m=n;
                while (--m) {
                        if (strcmp (*p, *(p+1)) > 0) {
                                t=*p; *p=*(p+1); *(p+1)=t;
                                gata=0;
                        }
```

```
                p++;
            }
        }
}
```

Un program de test este următorul (se folosește funcția **write_lines**, variantă corectă, definită în 20.6):

```
#include <stdio. h>
#include <string. h>
#define NMAX 10
char *x[NMAX];
void main (void)
{
        int n=read_lines (x, NMAX);
        write-lines (x, n);
        sort_lines (x, n);
        write_lines (x, n);
}
```

21. Este adecvat un tablou de dimensiune Nx4, definit prin:

```
#define N 10
float punct [N] [4];
```

Fiecare element punct [i] conține, în ordine, coordonatele spațiale x, y, z ale punctului respectiv și masa punctului. Calculul coordonatelor centrului de greutate se face cu secvența:

```
float xg, yg, zg, masa;
int i;
xg=yg=zg=masa=0.0;
for (i=0; i< N; i++) {
        xg+= punct[i][0] * punct[i][3];
        yg+= punct[i][1] * punct[i][3];
        xg+= punct[i][2] * punct[i][3];
        masa += punct [i][3];
}
xg /= masa;
yg /= masa;
zg /= masa;
```

22. Trebuie avut în vedere să nu se schimbe de două ori elementul de indici (i,j) cu elementul de indici (j,i), ceea ce ar lăsa matricea neschimbată. Se folosește funcția **schimbă** de la exercițiul 20.10.

```
void main (void)
{
        float a[5][5]= {
                    { 1,   2,   3,   4,   5},
                    { 6,   7,   8,   9,  10},
                    { 11, 12,  13,  14,  15},
                    { 16, 17,  18,  19,  20},
                    { 21, 22,  23,  24,  25}
        };
        int i, j;
        for (i=0; i< 5; i++)
```

```
            for (j=0; j < i; j++)
                    schimbă (&a[i][j]), &a[j][i], sizeof (float));
        for (i=0; i < 5; i++) {
            for (j=0; j < 5; j++)
                    printf ("%10.3f", a[i][j]);
            putchar ('\n');
        }
    }
```

25. Fişierele trebuie deschise cu funcţia **fopen**, care întoarce un pointer către un tip predefinit de date, tipul **FILE**. Deschiderea se va face pentru citire, respectiv scriere. Dacă se întoarce pointerul NULL, operaţia a condus la o eroare. Funcţiile **fgetc** şi **fputc** sunt analoge lui **getchar** şi **putchar**, dar citesc/scriu într-un fişier. Funcţia **fgetc** întoarce constanta întreagă predefinită **EOF** când se ajunge la sfârşit de fişier. Programul (numit **cp**) afişează şi o serie de mesaje de eroare.

```
#include <stdio.h>
void main (int argc, char *argv[ ])
{
    FILE *sursa, *dest;
    int c;
    if (argc !=3) {
            printf ("cp: Sintaxa este: cp sursa dest\n");
            exit(1);
    }
    if ((sursa=fopen (argv[1], "rb"))= =NULL) {
            printf ("cp: Eroare deschidere fişier %s\n", argv[1]);
            exit(1);
    }
    if ((dest =fopen (argv[2], "wb"))=NULL) {
            printf ("cp:Eroare deschidere fişier %s\n", argv [2]);
            exit (1);
    }
    while ((c=fgetc (sursa)) !=EOF)
            fputc (c, dest);
    fclose (sursa);
    fclose (dest);
}
```

26. Se declară o variabilă **a** de tip **float** $**$, căreia i se atribuie pointerul întors de **malloc**. Variabila **a** va conţine adresa matricii de dimensiune **n**. Pentru accesul la elementele matricii se foloseşte o variabilă **b** de tip **float** $*$, iniţializată cu $*$ **a** şi se simulează funcţia de alocare a tabloului. Citirea se face într-o variabilă temporară **x** şi apoi se depune în tablou valoarea citită, deoarece unele versiuni de compilatoare nu leagă formatele de tip float la funcţia **scanf** decât dacă sesizează explicit o variabilă de tip float.

```
#include <stdio.h>
#include <stdlib.h>
float **a;
void main (void)
{
```

```
	int i, j, n;
	float *b, x;
	printf ("Dimensiune matrice:");
	scanf ("%d", &n);
	a=(float **) malloc (n*n*sizeof (float));
	if (a==NULL) {
		printf ("Eroare alocare\n");
		exit (1);
	}
	b=*a;
	for (i=0; i < n; i++)
		for (j=0 ; j < n; j++) {
			scanf ("%f", &x);
			b[i*n+j]=x;
	}
	for (i=0; i < n; i++) {
		for (j=0; j < n; j++)
			printf ("%11.3f", b[i*n+j]);
		putchar ('\n');
	}
	putchar ('\n');
	free (a);
}
```

27.

```
void *my_calloc(size_t n, size_t size)
{
	size_t m=n * size;
	void *p;
	char *s;
	p = malloc (m);
	if (p !=NULL) {
		s=(char *)p;
		while (m--)
			*s++='\0';
	}
	return p;
}
```

30. Pentru calculul integralei definite, se defineşte o funcţie de forma:

```
#include <stdio. h>
#include <math. h>
#define NRVAL 100
double integrala (double (*f) (double), double a, double b)
}
	double pas=(b--a)/NRVAL;
	double x, suma=0.0;
	for (x = a; x < b; x+= pas)
		suma +=(*f)(x) * pas;
	return suma;
}
void main (void)
{
```

```
        double pi=4.0 * atan (1.0);
        double i;
        i = integrala (sin, 0.0, pi);
        printf ("Valoare teoretică= %f\n", 2.0);
        printf ("Valoare calculată= %f\n", i);
}
```

32

```
#define NREL (arr) (sizeof (arr) / sizeof (arr[0]))
#define SIZE (arr) (sizeof (arr[0]))
typedef unsigned char BYTE;
typedef int (*PFCMP) (const void *, const void *);
int lin_1 (void *key, void *base, size_t n, size_t size, PFCMP cmp)
{
        int i;
        BYTE *a=(BYTE *) base;
        for (i=0; i < n; i++)
                if ((*cmp) (key, a+i*size)==0)
                return i;
        return —1;
}
```

33. Pentru interschimbare se folosește funcția **schimbă** din exercițiul 20.10. Elementul găsit se schimbă cu cel din stânga sa numai dacă indicele său nu este 0 (deci dacă există un element al tabloului la stânga sa). Un program complet de test este:

```
#include <stdio. h>
void schimbă (void *a, void *b, size_t n)
{
        char temp;
        while (n— —) {
                temp=*((char *)a);
                *((char *)a)++=*((char *)b);
                *((char *)b)++=temp;
        }
}
int lin_2 (void *key, void *base, size_t n, size_t size, PFCMP cmp)
{
        int i;
        BYTE *a=(BYTE *) base:
        for (i=0; i < n; i++)
              if ((*cmp) (key, a+i*size)==0) {
                          if (i > 0) {
                                    schimbă (a+(i—1)*size. a+i*size, size);
                                    return i—1;
                          }
                          else
                                    return i;
              }
        return —1;
}
int cmp_num (const void *a, const void *b)
{
```

```
        return (*(int *)a — *(int *)b);
}
int tab [10]={1, 2, 4, 6, 7, 10, 12, 14, 21, 34};
void main (void)
{
        int i;
        int key;
        key=4;
        i=lin_2( &key, tab, NREL (tab), SIZE (tab), cmp_num);
        printf ("%d\n", i);
        i=lin_2( &key, tab, NREL (tab), SIZE (tab), cmp_num);
        printf ("%d\n", i);
}
```

Capitolul 21

3.

```
typedef struct {float mod; float arg;} POLAR;
typedef struct {float re; float im;} COMPLEX;
POLAR conv_polar (COMPLEX z)
{
        POLAR w;
        w. mod=sqrt (z.re * z. re+z. im * z. im);
        w. arg=atan2(z. im, z. re);
        return w;
}
COMPLEX conv_complex (POLAR w)
{
        COMPLEX z;
        z. re=w. mod * cos (w. arg);
        z. im=w. mod * sin(w. arg);
        return z;
}
```

4.

```
COMPLEX z[3];
float perim=0.0;
int i, j;
float sqr (float x)
{
        return x*x;
}
for (i=0; i< 3; i++) {
        int (j=(i+1) %3;
        perim += sqrt (sqr(z[i].re—z[j].re) + sqr(z[i].im—z[j].im));
}
```

7.

```
COMPLEX z[10], *pz=z;
while (pz—z < 10) {
        printf ("%f+i * %f\n", pz—>re, pz—>im);
        pz++;
}
```

11. Ridicarea la putere a unui număr în formă algebrică se poate face cu ajutorul conversiei în formă trigonometrică şi al utilizării formulei lui Moivre. Se folosesc structurile şi funcţiile de la exerciţiul 21.3

```
#include <math.h>
typedef struct {float mod; float arg; } POLAR;
typedef struct {float re; float im;} COMPLEX;
float ipower (float x, int n)
{
        float p=1.0;
        if (n< 0)
              return 1.0/ipower (x, —n);
        else if (n==0)
                    return 1.0;
              else {
                    while (n— —)
                          p *=x;
                    return p;
              }
{
COMPLEX cpower (COMPLEX z, int n)
{
        POLAR w=conv_polar (z);
        w. mod=ipower (w. mod, n);
        w. arg *=n;
        return conv_complex (w);
}
```

12.

```
typedef struct {float x; float y; float z;} VECTOR;
typedef float SCALAR;
VECTOR prod_vec_scal (VECTOR v, SCALAR a)
{
        v.x*=a;
        v.y*=a;
        v.z*=a;
        return v;
}
SCALAR prod_scal (VECTOR v1, VECTOR v2)
{
        return v1. x * v2. x+v1. y * v2. y+v1. z*v2. z;
}
VECTOR prod_vec (VECTOR v1, VECTOR v2)
{
        VECTOR w;
        w. x=v1. y * v2. z—v1. z * v2. y;
        w. y=v1. z * v2. y—v1. y * v2. z;
        w. z=v1. x * v2. y—v1. y * v2. x;
        return w;
}
```

13.

```
typedef struct {float x; float y;} PUNCT;
typedef struct {PUNCT centru; float raza;} CERC;
```

```
float sqr (float x)
{
        return x*x;
}
float dist2 (PUNCT p1, PUNCT p2)
{
        return sqr (p1.x— p2.x)+sqr(p1.y—p2.y)
}
float putere (PUNCT p, CERC c)
{
        return dist2 (p, c. centru) — sqr (c. raza);
}
```

14.

```
void print_invers (LINK t)
{
        if (t==NULL)
                printf ("NULL");
        else {
                print_invers (t— >next);
                printf ("<— —");
                putchar (t— >d);
        }
}
```

15.

```
int list_len (LINK t)
{
        int n=0;
        while (t !=NULL) {
                t=t— >next;
                n++;
        }
        return n;
}
int list_len (LINK t)
{
        if (t==NULL)
                return 0;
        else
                return 1+list_len (t— >next);
}
```

16. Se folosește funcția new_el().

```
LINK list_dup (LINK t)
{
        LINK p;
        if (t==NULL)
                return NULL;
        else {
                p=new_el (t— >d, NULL);
                p— >next=list_dup(t— >next);
```

```
        return p;
    }
}
```

17. Se parcurge prima listă până la ultimul element și apoi se leagă acest ultim element cu o copie a celei de-a doua liste, obținută cu funcția list_dup din exercițiul anterior.

```
void list_cat (LINK t, LINK p)
{
    if (t==NULL) {
        printf ("list_cat: Prima listă vidă\n");
        exit (1),
    }
    while (t-->next !=NULL)
        t=t-->next;
    t-->next=list_dup (p);
}
```

Trebuie observat testul din bucla de parcurgere a primei liste și atribuirea respectivă. Este **greșită** o construcție de forma:

```
while (t !=NULL)
    t=t-->next;
t = list_dup (p);
```

deoarece, datorită transferului prin valoare, modificarea variabilei t (parametru formal al funcției) nu se reflectă în afara acesteia. Atribuirea:

```
t-->next=list_dup (p);
```

echivalentă prin definiție cu:

```
(*t). next=list_dup (p);
```

accesează câmpul **next** prin intermediul pointerului **t**, în felul acesta realizându-se un transfer prin referință **și**, deci, modificarea elementului indicat de **t**.

18. Problema constă în a defini funcții de comparație adecvate.

```
typedef int (*PFCMP) (const void *a, const void *b);
int cmp_1 (const void *a, const void *b)
{
    struct elev *pa=(struct elev *)a;
    struct elev *pb=(struct elev *)b;
    return strcmp (pa-->nume, pb-->nume);
}
int cmp_2 (const void *a, const void *b)
{
    struct elev *pa=(struct elev *)a;
    struct elev *pb=(struct elev *)b;
    return pa-->an — pb-->an;
}
```

Sortarea propriu-zisă se realizează prin apelul funcției **qsort**.

```
struct elev tab [20];
qsort (tab, 20, sizeof (struct elev), cmp_1);
qsort (tab, 20, sizeof (struct elev), cmp_2);
```

20.

```
union ud {double x; char c[8];};
void hex_print (double x)
{
        union ud u;
        int i;
        u.x=x;
        for (i=7; i>0; i— —)
                printf ("%02x", ud. c[i]);
}
```

21. Se definește o structură cu câmpuri de biți care să respecte configurația registrului de flaguri și o uniune în care se suprapun o asemenea structură și un întreg pe 16 biți.

```
#include <stdio. h>
typedef unsigned short WORD;
typedef union {
        struct {
                WORD _cf:1;
                WORD    :1;
                WORD _pf:1;
                WORD    :1;
                WORD _af:1;
                WORD    :1;
                WORD _zf:1;
                WORD _sf:1;
                WORD _tf:1;
                WORD _if:1;
                WORD _df:1;
                WORD _of:1;
                WORD    :4;
        }f;
        WORD w;
} FLAGS;
FLAGS f;
WORD get_flags (void)
{
        asm {
                pushf;
                pushf;
                pop ax;
        }
        _AX &=0x0fd5;
        asm popf;
        return _AX;
}
void type_flags (void)
{
        asm pushf;
```

```
        f.w=get_flags ( );
        printf ("OF= %d DF= %d IF= %d TF= %d SF= %d",
                f.f._of, f.f._df, f.f._if, f.f._tf, f.f._sf);
        printf ("ZF= %d AF= %d PF= %d CF= %d\n",
                f.f._zf, f.f._af, f.f._pf, f.f._cf);
        asm popf;
}
```

Capitolul 22

3.

```
int read_tab(int a[ ], int n, char *fisier)
{
        int i = 0, m, x;
        FILE *fp=fopen (fișier, "rb");
        if (fp==NULL) {
                fprintf (stderr, "read_tab: eroare deschidere\n");
                exit (1);
        }
        while (i<n) {
                m=fread (&x, sizeof i(nt), 1, fp);
                if (m > 0)
                        a[i++]=x;
                else
                        break;
        }
        fclose (fp);
        return i;
}
```

Funcția **write_tab** se scrie într-o manieră similară.

5. Programul face afișarea conținutului unui fișier la consolă în diferite formate (ASCII, în hexa, sau în octal). Programul (numit **dump**) primește ca argumente numele fișierului (care poate conține „wildcards", adică * și ?) și eventuale opțiuni (**—a** sau **—x** sau **—o**). În program se testează corectitudinea argumentelor (număr, etc.).

```
#include <stdio. h>
#include <dir. h>
#include <string. h>
#include <ctype. h>
typedef unsigned char BYTE;
static BYTE s[80];
static char *help="Sintaxa este: dump <file> [— a|—x|—o]\n";
static char *error="dump: Eroare deschidere fișiere %s. . \n";
static size_t nr_bytes=80;
static nr_blank=0;
void prints (BYTE *s, int n, int baza, unsigned count)
{
        int i;
        char *fmt;
        if (n==0)
                return;
```

```
        if (!(baza==16 || baza==8 || baza=='A'))
                return;
        switch (baza) {
                case 16 : fmt="%02X"; break;
                case  8 : fmt="%03o"; break;
                case 'A':
                default : fmt="%c"; break;
        }
        if (baza !='A')
                if (baza ==8)
                        printf ("%06o", count);
                else
                        printf ("%04X", count);
        for (i=0; i< n; i++)
                printf (fmt, s[i]);
        if (baza=='A')
                return;
        switch (baza) {
                case 16 :   i=(nr_bytes—n)*3; break;
                case  8 :   i=(nr_bytes—n)*4; break;
                default :   break;
        }
        while (i——)
                putchar (' ');
        for (i=0; i< nr_blank; i++)
                putchar (' ');
        for (; n—— >0; s++) {
                if (isprint (*s))
                        putchar (*s);
                else
                        putchar ('.');
        }
        putchar ('\n');
    }
    void prel_file (char *fname, int baza)
    {
        sizef_t     m;
        FILE        *fp;
        unsigned count =0;
        if ((fp=fopen (fname, "rb"))==NULL) {
                printf (error, fname);
                return;
        }
        printf ("\nFile: %s\n\n", strlwr (fname));
        do {
                m=fread (&s, 1, nr_bytes, fp);
                if (m==0)
                        break;
                prints (s, m, baza, count);
                count += m;
        } while (m==nr_bytes);
        fclose (fp);
```

```
}
voaid main (int argc, char **argv)
{
	int   baza, done;
	struct ffblk work;
	baza='A';
	switch (argc) {
		case 2: baza ='A'; break;
		case 3: if (strcmp (argv[2], "—x")==0)
				baza=16;
			else
				if (strcmp argv[2], "—o")==0)
					baza=8;
				else
					if (strcmp (argv[2], "—a")==0)
						baza='A';
					else {
						printf (help);
						exit (1);
					}
			break;
		case 1:
		default: printf (help);
			exit (1); break;
	}
	if (baza !='A') {
		nr_bytes=baza;
		nr_blank=(baza==8) ? 80—14—nr_bytes * 5:
			80—8—nr_bytes * 4;
	}
	done=findfirst (argv[1], &work, 0);
	while (!done) {
		prel_file (work.ff_name, baza);
		done=findnext ( &work);
	}
}
```

Afişarea se poate face în două moduri: ASCII (opţiunea —a), în care se tipăresc caracterele citite din fişier (are sens numai pentru fişiere text) şi în hexazecimal sau octal (opţiune —x sau —0), în care se tipăresc valorile octeţilor citiţi din fişier şi, în marginea din dreapta a ecranului, reprezentarea lor ASCII (dacă sunt caractere netipăribile, se afişează un punct).

Fişierul header **ctype.h** este inclus pentru a folosi **isprint()**, care testează dacă un caracter este tipăribil (caracterele ASCII tipăribile au codurile între 0x20 şi 0x7e inclusiv).

Variabila **nr_bytes** va indica numărul de octeţi ce se vor afişa pe o linie în cazul hexa sau octal (16 şi, respectiv, 8) iar variabila **nr_blank** va indica numărul de blank-uri care se lasă pe o linie între reprezentarea în **hexa** sau octal şi cea ASCII corespunzătoare. Acest mod de tratare e necesar deoarece un octet se reprezintă pe 2 cifre în **hexa** şi pe 3 cifre în octal.

Funcţia **prints**, care afişează o linie pe ecran, primeşte un tablou **s** de caractere (fără terminatorul '\0' la coadă), numărul **n** de caractere din tablou, un întreg **baza** care poate avea valorile 'A', 8 sau 16 (reprezentând modul

de afișare) și un întreg fără semn **count** care (în cazul **baza** ==8 sau 16) va preciza numărul de octeți afișați curent; acest număr se va tipări la începutul unei linii. După câteva verificări (dacă **n**==0, nu avem ce tipări, de asemenea dacă **baza** nu este corect, nu se face nimic), se selectează formatul corespunzător pentru funcția **printf**, care poate fi „%**c**", „%**02X**" sau „%**03o**", adică tipărire caracter, întreg hexa pe două cifre sau întreg octal pe 3 cifre, în ultimele două cazuri completându-se formatul cu zero-uri la stânga. Se tipărește **count** și apoi elementele tabloului **s**, cu formatul ales.

Dacă suntem în cazul ASCII, totul este O.K. Dacă nu, se calculează câți octeți s-au reprezentat în linia curentă; nu e obligatoriu ca acest număr să fie egal cu **nr_bytes** (în ultima linie s-ar putea să avem mai puțini octeți). Dacă acest număr de octeți este mai mic decât **nr_bytes**, se tipărește un număr de blank-uri egal cu diferența lor, apoi **nr_blank** blank-uri. În fine se tipărește reprezentarea ASCII a celor **n** octeți afișați (dacă sunt tipăribili) sau un punct pentru cei netipăribili. Înainte de return, se sare la linie nouă.

Funcția **prel_file**, responsabilă cu afișarea unui fișier, primește numele fișierului și baza de afișare. Se deschide pentru citire în modul binar fișierul specificat, prin funcția **fopen**, care întoarce un pointer **fp** către tipul de date **FILE** sau **NULL** dacă nu s-a făcut corect deschiderea. În cazul incorect, se tipărește mesajul de eroare și se revine în programul apelant. Se tipărește numele fișierului, convertit cu funcția **strlwr** la litere mici. Începe apoi citirea din fișier. Funcția **fread** citește **nr_bytes** înregistări din fișierul **fp**, toate având lungimea 1 octet, depunând datele citite în bufferul **s**. Funcția întoarce numărul de înregistări citite (în cazul de față chiar numărul de octeți), care poate fi mai mic strict decât **nr_bytes** sau chiar 0, la ultima citire. Numărul de octeți efectiv citiți este transmis (dacă nu e 0) către funcția **prints**, împreună cu adresa bufferului **s**, cu baza aleasă și cu contorul de octeți. Funcția **prints** va afișa corespunzător linia pe ecran. După apelul lui **prints** se actualizează **count**, care fusese inițializat cu **0**. Toată această operație se desfășoară într-un ciclu cu test la partea inferioară, din care se iese dacă numărul de octeți citiți efectiv este mai mic strict decât **nr_bytes** sau este **0**. În final se închide fișierul **fp** și se revine în programul apelant.

Se definește la nivelul exterior al programului un buffer s de lungime 80, care va memora numărul de octeți care se afișează pe o linie. Se mai definesc două mesaje, unul care să fie tipărit dacă argumentele nu sunt corecte, iar unul dacă nu se poate deschide fișierul (de exemplu s-a dat greșit numele).

Programul principal testează întâi numărul de argumente **argc**. O lansare corectă în execuție poate fi cu **argc**==**2** sau **argc**==**3**. Dacă **argc** este 2 (adică singurul argument propriu-zis, este numele fișierului), se poziționează **baza** la valoarea **'A'** (cazul implicit). Dacă **argc** este 3, înseamnă că s-a dat o opțiune și se poziționează **baza** la **'A'**, **8** sau **16**, funcție de opțiune (opțiunea este în **argv[2]**). Dacă opțiunea nu este corectă sau dacă **argc** nu este **2** sau **3**, se tipărește mesajul **help** și programul se termină. Testarea opțiunii se face cu **strcmp()**.

Se calculează apoi numărul de octeți **nr_bytes** care se citesc curent din fișier; acesta depinde de baza aleasă și este 80, 16 sau 8. De asemenea se calculează **nr_blank**.

Deoarece numele fișierului poate conține caracterele generice * și ?, trebuie accesat directorul curent pentru a prelua toate fișierele care corespund numelui generic specificat. Structura cu numele **work**, de tip **ffblk** (tip predefinit), este adecvată acestei operații. Funcția **findfirst()**, cu parametri numele fișierului, adresa structurii de tip **ffblk** și atributele fișierului (uzual 0) va întoarce valoarea **0** dacă a găsit un fișier cu nume adecvat sau −1 în caz

contrar. Dacă s-a găsit un fişier, numele său este disponibil în tabloul de caractere **work.ff_name** din structura **work.** Se apelează deci **prel_file()** cu acest nume şi apoi se apelează **findnext()** pentru a identifica alte fişiere care corespund numelui generic specificat. Operaţia se repetă până când **findnext()** întoarce −1, semnalând că nu sunt fişiere de prelucrat.

Exemple de folosire ale programului **dump:**

⟩dump *.c⟨CR⟩

(se vor afişa toate fişierele cu extensia .c în ASCII)

⟩dump t*.* −o ⟨CR⟩

(se vor afişa toate fişierele al căror nume începe cu t, cu toate extensiile, în octal)

6. Se foloseşte aceeaşi structură generală de program ca în exerciţiul precedent, împreună cu o funcţie de prelucrare adecvată (conversie şir de caractere din litere mici în litere mari).

7. Se foloseşte aceeaşi structură generală de program ca în exerciţiul 5) împreună cu o serie de contoare care se incrementează funcţie de caracterul curent citit din fişier.

8. Problema se reduce la a număra cuvintele dintr-un şir dat de caractere. Şirurile introduse (cel mult 20) se salvează cu **strdup()** şi adresele lor se depun într-un tablou.

```
#include <stdio.h>
#include <string.h>
char *delim="\t\n";
void list_words (char *s)
{
        char *p;
        p=strtok (s, delim);
        while (p !=NULL) {
                puts (p);
                p=strtok (NULL, delim);
        }
}
char *t[10];
void main (void)
{
        char buf[80];
        int i, n;
        for (n=0; n < 10; n++) {
                gets (buf);
                if (buf[0]=='\0')
                        break;
                else
                        t[n]=strdup (buf);
        }
        for (i=0; i < n; i++)
                list_words (t[i]);
}
```

Şirul constant **delim** poate include şi alţi delimitatori, cum ar fi punctul, virgula, etc.

9. Şirul dat nu conţine spaţii albe, deci poate fi interpretat ca un cuvânt. Se citesc linii de text (cu funcţia **fgets**) din fişierul text specificat şi se de-

limitează textul citit (cu **strtok**), considerând ca delimitatori spațiile albe. Se compară fiecare subșir raportat de funcția **strtok** cu șirul dat (comparația se face cu **strcpy**) și se contorizează fiecare coincidență.

10. Problema se rezolvă similar cu cea precedentă, scriind în al doilea fișier textul modificat: ori de câte ori apare primul șir, se va scrie în loc cel de-al doilea șir. Trebuie ținute două copii ale șirului scanat cu funcția **strtok** deoarece aceasta scrie '\0' în poziția curentă identificată din șir, distrugând astfel un delimitator. Se poate memora însă poziția întoarsă de **strtok** și se poate avea acces și la delimitatorii pe care **strtok** îi sare în mod inerent.

12. Se poate folosi funcția **strtol**, dar trebuie scrisă o funcție cu semnificație inversă, care să genereze un șir de cifre într-o bază dată, pornind de la o valoare numerică.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void ltoasc (long n, char *s, int baza)
{
        long sign;
        unsigned long m;
        int v;
        char *p=s;
        if (baza < 2|| baza > 36) {
                printf ("Baza incorectă\n");
                exit (1);
        }
        if (sign=n < 0)
                m=—n;
        else
                m=n;
        do {
                v=m%baza;
                *p++=v+((v > 9) ? 'A' : '0');
        } while ((m/=baza) !=0);
        if (sign < 0)
                *p++='—'.
         *p='\0';
        strrev (s);
}
void conv_rad (char *sursa, char *dest, int b_sursa, int b_dest)
{
        long n;
        if (b_sursa < 2|| b_sursa > 36) {
                printf ("Baza incorectă\n");
                exit (1);
        }
        if (b_dest < 2|| b_dest > 36) {
                printf ("Baza incorectă\n");
                exit (1);
}
```

```
    n=strtol (sursa, NULL, b_sursa);
    ltoasc (n, dest, b_dest);
}
```

15. Pentru a discerne între primul apel și cele ulterioare, se folosește un flag declarat static. Funcția **timp** testează acest flag și, dacă îl găsește **0** (adică este primul apel), îi dă valoarea 1. Pentru ca la încheierea programului să se execute aceeași funcție indiferent de locul și modul de ieșire (cu funcția **exit**, prin încheiere normală etc.), se folosește funcția **atexit**. Programul principal va începe cu o secvență de forma:

```
if (atexit (timp)) {
        printf ("Eroare atexit\n");
        exit ( );
}
timp ();
```

Funcția **timp()** este prezentată în continuare:

```
#include <stdio.h>
#include <time.h>
#include <stdlib.h>
static clock_t tim;
unsigned long zec;      /*Zecimi de secundă*/
int flag=0;
void timp (void)
{
      if (flag==0) {
            tim=clock ( );
            flag=1;
            return;
      }
      tim=clock ( )—tim;
      zec=(tim *55)/100;
      printf ("Timp execuţie: %2lu. %01lu secunde", zec/10, zec %10);
}
```

16.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NREL(a) (sizeof (a)/sizeof (a[0]))
typedef struct my_time {int luna; int zi; char *name;} MY_TIME;
MY_TIME calendar [ ]= {
      {5, 10, "Ionescu"},
      {5, 31, "Marinescu"},
      {7, 9, "Iordănescu"},
      {4, 13, "Mihăilescu"},
      {5, 31, "Popescu"}
};
int cmp (MY_TIME a, MY_TIME b)
{
      return a.zi==b.zi && a. luna==b. luna;
}
```

```
void main (void)
{
        struct tm *timp;
        time_t t;
        MY_TIME zi;
        int i;
        time (&t);
        timp=localtime (&t);
        zi. luna=timp->tm_mon+1;
        zi.zi=timp -> tm_day;
        zi.name=NULL;
        for (i=0; i< NREL (calendar); i++)
                if (cmp (zi, calendar [i]))
                        printf ("Astăzi este ziua de naştere a lui %s\n",
                                calendar [i]. name);
}
```

BIBLIOGRAFIE

1. Al. Kelley, Ira Pohl *A Book On C.* The Benjamin/Cummings Publishing Company.
2. Brian Kernighan, Dennis Ritchie *The C Programming Language* Prentice Hall, 1978, 1988.
3. Ciocîrlie H., Ilieş P., Balla I. *Limbajele de programare PASCAL şi PASCAL concurent* Editura Facla, 1985.
4. Cristea V., Athanasiu I., Kalisz E., Panoiu A. *Turbo Pascal 60.* Editura Teora, Bucureşti, 1992.
5. Davidovici A., Bărbat B. *Limbaje de programare pentru sisteme în timp real* Editura Tehnică, Bucureşti, 1986.
6. Emmet Beam *Illustrated C Programming* ANSI Standard Wordware Publishing Inc., 1989.
7. Findlay W., Watt D. *PASCAL, An Introduction To Methodical Programming* Pitman Publishing, 1987.
8. Ghezzi C., Jazaveri M. *Programming Language Concepts* John Willey & Sons, 1987.
9. Herbert Schildt *C: The Complete Reference* Mc. Graw Hill, 1990.
10. Horowitz E., Sahni S. *Fundamentals of Data Structures in PASCAL* Computer Science Press, 1976.
11. Iorga V., Fătu I. *Programarea în limbajul Pascal* Litografia I.P.B., 1987.
12. Jensen K., Wirth N. *PASCAL, User Manual and Report* Springer-Verlag, 1975.
13. Kierburtz R. *Structured Programming and Problem-Solving with PASCAL* Computer Science Press, 1976.
14. Les Hancock, Morris Krieger *The C Primer* Mc. Graw-Hill Book Company, 1986.
15. Robert Lafore. *Turbo C Programming For The IBM* Howard W. Sams & Company, 1987.
16. Şerbănaţi L. D. *Limbaje şi compilatoare. Programare în Pascal* Litografia I.P.B., 1983
17. Şerbănaţi L,. D. *Limbaje de programare şi compilatoare* Editura Academiei, Bucureşti, 1987.
18 Şerbănaţi L. D., Moldoveanu F., Iorga V., Valeriu P. *Programarea sistematică în limbajele FORTRAN şi PASCAL* Editura Tehnică, Bucureşti, 1984.
19. Yourdon E., Constantine L. *Structured Design* Prentice Hall, 1979.
20. . . . . *Turbo C — Manual de utilizare*
21. . . . . *Turbo C++ — Manual de utilizare*
22. . . . . *Borland C++ — Manual de utilizare*

CUPRINS

PARTEA I. **Limbajul de programare PASCAL**

PARTEA A II-A. **Limbajul de programare C**